SEER: Auto-Generating Information Extraction Rules from User-Specified Examples

Maeda F. Hanafi, Azza Abouzied
New York University - Abu Dhabi
{maeda.hanafi, azza}@nyu.edu

Laura Chiticariu, Yunyao Li
IBM Research - Almaden
{chiti, yunyaoli}@us.ibm.com

ABSTRACT

Time-consuming and *complicated* best describe the current state of the Information Extraction (IE) field. Machine learning approaches to IE require large collections of labeled datasets that are difficult to create and use obscure mathematical models, occasionally returning unwanted results that are unexplainable. Rule-based approaches, while resulting in easy-to-understand IE rules, are still time-consuming and labor-intensive. SEER combines the best of these two approaches: a learning model for IE rules based on a small number of user-specified examples. In this paper, we explain the design behind SEER and present a user study comparing our system against a commercially available tool in which users create IE rules manually. Our results show that SEER helps users complete text extraction tasks more quickly, as well as more accurately.

ACM Classification Keywords

H.5.m. Information Interfaces and Presentation (e.g. HCI): Miscellaneous

Author Keywords

Data Extraction; Example-driven learning

INTRODUCTION

With the growing amount of unstructured and semi-structured text, Information Extraction (IE) is becoming increasingly important. IE is at the core of many emerging applications, such as eDiscovery, brand management based on social media, and risk analysis using financial reports. Despite significant advances in IE in recent years, IE solutions continue to be expensive. Users either have to train machine learning models with large labeled datasets or manually develop extraction rules in a scripting language such as Python or Perl, or a special data extraction language. In particular, while the rule-based approach is widely used in practice for its explainability and maintainability [9], developing IE rules is known to be time-consuming and labor-intensive.

To illustrate, suppose you are an analyst given a set of documents to extract financial revenue information. As a novice developer, you spend hours learning and debugging code in Python. Finally, your script reads each document and prints every instance at which the dollar sign appears. As you analyze the extractions and compile varying target extractions, you iteratively improve the extraction results, e.g. keep only text that contains financial terms such as 'revenue' or 'annual income', until you achieve the desired quality of results.

SEER addresses the key challenges in the development cycle of IE scripts or rules by (1) reducing the barrier of learning a new programming language for novice developers, and (2) minimizing the time-consuming, manual effort required in constructing rules. SEER is an IE tool that suggests easy-to-understand extraction rules from a small set of extracted-text examples provided by users. Users provide examples by highlighting text from documents that they wish to extract. Based on the highlighted examples, SEER learns extraction rules in Visual Annotation Query Language (VAQL) [20]. VAQL is a commercially available declarative graphical language for IE rule development that is simple, yet powerful and expressive. It consists of *primitives* that capture tokens of a pre-defined concept (e.g., telephone numbers, names or organizations), or that match a regular expression or a literal, among others, as well as constructs for combining these primitives.

SEER's learning algorithm only requires a few examples to correctly suggest rules; in addition, it ensures that the suggested rules are *diverse* in structure: they contain different primitives. Thus, SEER's learning algorithm is necessarily *heuristic* in nature; heuristics are used to encode and leverage domain knowledge that would otherwise be unavailable to the algorithm in the absence of a large labeled dataset. For example, by observing rule developers constructing rules in VAQL, we noticed that certain primitives are preferable to others. For example, most users prefer a rule with a *pre-built primitive* that captures telephone numbers to a rule with multiple *regular expression primitives* that capture different parts of a telephone number. SEER's heuristic algorithm utilizes this natural preference ordering of primitives to rank its suggested rules. SEER is thus a sweet spot between data-hungry machine learning techniques (including probabilistic rule generation methods) that require large labeled datasets to ensure robustness [28] and manual rule development systems like VINERy [20]. With SEER, we demonstrate that heuristics are a powerful method for human-in-the-loop synthesis systems of extraction rules; they help guide and minimize the search space, turning only a few data examples into a small set of diverse rules with high accuracy and coverage.

CHI 2017, May 6-11, 2017, Denver, CO, USA.

http://dx.doi.org/10.1145/3025453.3025540

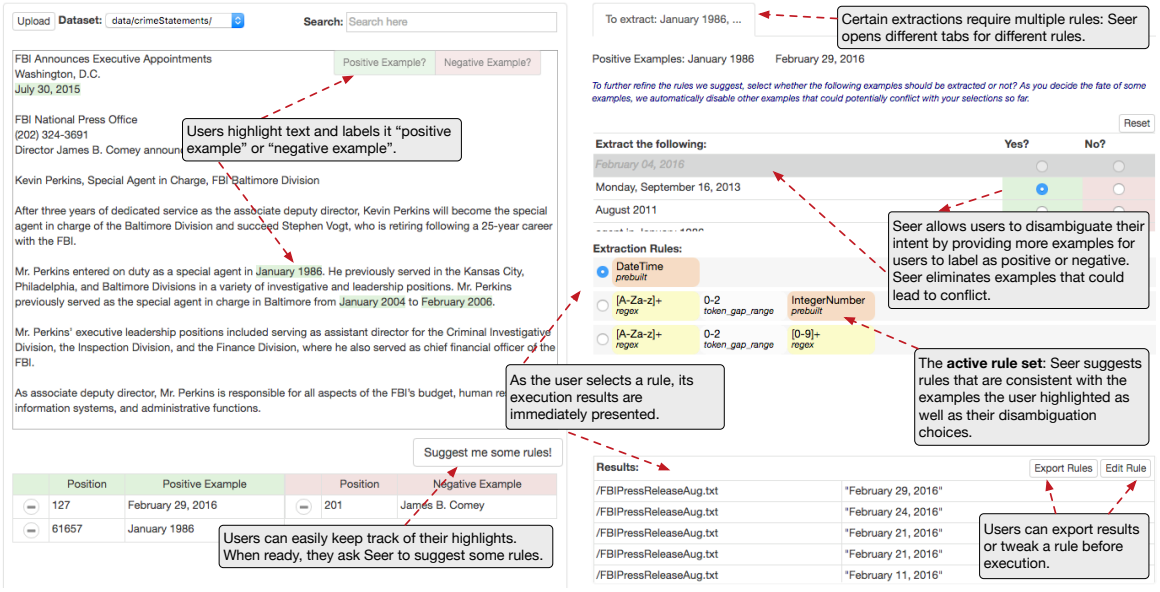


Figure 1. SEER's Interface

In general, users are more effective at making a selection when presented with fewer options to choose from. Therefore, alongside the initial set of suggested rules, SEER displays a collection of extractions, known as *refinements*, that help differentiate the suggested rules. Based on user feedback (accept or reject) on these refinements, SEER further refines the set of suggested rules.

We compare SEER's automatic rule suggestion to manual rule creation in VINERy [20], a commercially available drag-and-drop interface for VAQL extraction rules. Our results show that users complete text extraction tasks more quickly and accurately with SEER. By requiring examples as input, SEER reinforces a more thorough approach to IE: users closely inspect the dataset, provide more extraction variations and verify the correctness and completeness of an extraction rule.

Outline. In this paper, we give an overview of SEER, and present the heuristic learning algorithm, the ranking of VAQL primitives, and the refinement suggestion algorithm. We then present our user study results, and discuss prior work in IE.

UI OVERVIEW AND WALKTHROUGH

We explain how a user, Bob, can use SEER to construct an extraction rule that extracts dates (e.g. January 29, 1999) from a data set of press releases. Bob performs the following steps:

1. Highlights Examples: Bob begins by analyzing his data, shown on the left-hand side of the interface (Figure 1). Since Bob wishes to extract dates, he searches for the months and highlights an example of a date and SEER colors it with a yellow highlight. Bob then indicates whether it is positive or negative. The highlighted text will be added in the tables under the document. After Bob finishes highlighting the examples 'February 29, 2016' and 'January 1986' as positive examples and 'James B. Comey' as a negative example, Bob clicks the "Suggest me some rules!" button. SEER requires at least one positive example before it can start learning rules.

2. Refines Rules and Analyzes Results: Once SEER learns rules consistent with the examples, the right-hand side of the interface (Figure 1) shows: (1) a list of *refinements* and (2) a list of suggested rules. Refinements are extraction results of the suggested rules that the user can accept or reject to further filter the suggested rules. Accepting and rejecting extractions is a quicker alternative to manually analyzing each rule. After a few refinements, Bob is left with a few rules. Bob selects one of the rules, and SEER executes the rule. The extraction results (called *extractions*) appear in the results table on the bottom right panel, and are highlighted in green in the document view on the left-hand side. Bob can analyze and verify the extractions by scrolling through the results table or the documents. If Bob is not satisfied with any of the suggested rules, he can add more examples or edit one of the rules directly.

PRELIMINARIES

Before we describe how SEER infers rules from examples, we will first define terminologies used throughout the paper.

Underlying Language of SEER

SEER's rules are specified using a subset of Visual Annotation Query Language (VAQL) [20], and are executed with the SystemT engine [16]. When given user-provided text examples, SEER infers rules in VAQL and suggests the easy-to-understand rules, in graphical form, to the user.

Tokenizer

Rules extract text from documents pre-tokenized by VAQL. VAQL delimits text on whitespace characters, such as spaces, newlines, tabs, etc.. VAQL considers symbols like dashes, commas, etc., as tokens. Hence, '1998-Jan14' would have the following tokens: '1998', '-', and 'Jan14'. VAQL also has a multilingual tokenizer that handles more complex delimiters. However, non-English text is outside the scope of this paper.

Primitives and Rules

SEER learns a disjunctive union of VAQL sequence rules, where each rule consists of a sequence of one or more extraction *primitives*, of one of the types listed below. The results of executing a rule or primitive over the input text are called *extractions*. We say that a rule (or primitive) *captures* a token (or a sequence of tokens) if the token (sequence of tokens) is among the extractions of the rule (primitive) on the input text.

- **Pre-built:** Pre-builts capture the tokens (one or more) of a particular concept such as organization, person, phone number, etc. For example, the pre-built P: Percentage captures percentages like '8 percent' and '50.5 percent'. Pre-builts can capture multiple tokens.
- **Literal:** Literals capture one or more tokens matching an exact string. For example, the literal L: 'percent' captures all tokens 'percent' that appear in the text.
- **Dictionary** Dictionaries contain a set of literals and capture tokens that match one of those literals. For example, the D: {percent, dollar} captures any of the two words that appear in the text.
- **Token Gap**: Token gaps skip over a number of tokens. Token gaps cannot be placed in the beginning or end of a rule. Within the rule, L: '50' T: 0-3 L: 'percent', the token gap T: 0-3 skips over zero to three tokens to capture the following: '50.5 percent' or '50 percent'.
- **Regular Expression**: Regular expressions can be any one of the following: R: [A-Za-z]+ captures a token consisting of letters, R: [0-9]+ captures a token consisting of digits, and finally R: symbols captures tokens consisting of symbols such as $, !, #, etc. One can easily extend the base regular expressions.

Users may supply their own pre-builts in addition to the default pre-builts in VAQL. Such pre-builts can be rules from other primitives. For instance, a user can import a pre-built for ordinal numbers, P: Ordinal Numbers, which he created from a dictionary primitive, D: {first, second, third}.

SEER can execute multiple VAQL rules, consequently permitting overlapping extractions captured by different rules. For instance, a rule capturing percentages may have overlapping extractions with a rule that captures numbers, e.g. '50.5 percent' overlaps with '50.5'. While VAQL allows the user to manually add a consolidation to the rules to combine the overlapping extractions, VAQL does not impose the requirement of disambiguating between the extractions. The output of executing multiple rules is a disjunction of all the extractions. Learning rule consolidations is outside the scope of this paper.

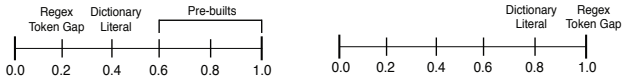
Primitive Scores

Scores are assigned to primitives and rules in order to prune the search space of feasible rules by eliminating low-scoring rules. Our scoring function is based on our hypothesis that rule developers prefer certain types of primitives for different types of tokens. We, therefore, assign scores to primitives based on this preference ordering. In the results section, we show that our hypothesized ranking of primitives is supported by the results. The score of a primitive is influenced by the semantics and syntactic properties of the token.

Semantic and Syntactic Tokens

There are two types of tokens, **semantic tokens** and **syntactic tokens**. A token is semantic if: (1) it can be captured by a pre-built, or (2) it appears more than once across the positive examples and does not appear in the negative examples. A token is syntactic if it is not semantic.

For each token type, SEER defines two different partial orders of the primitive types in order to quantify developer default preferences. If a primitive's token is semantic, then its score is obtained from the semantic scale; otherwise, its score is obtained from the syntactic scale. The scales are shown in Figure 2. Primitives are shifted to the end of the syntactic scale to make scores comparable to that on the semantic scale.



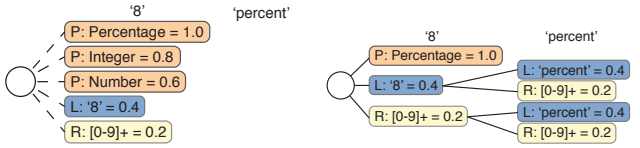
(a) Semantic Scale (b) Syntactic Scale

Figure 2. Scales for semantic and syntactic tokens. Note that the pre-builts cover a range, since there are ranks within the different concepts.

In the semantic scale, pre-builts are ranked the highest because we hypothesize that rule developers prefer pre-builts over other primitives. To illustrate, suppose the task is to extract percentages and associated dates, and the target extractions include '50.5 percent in September 2004' and '8 percent during December 2012'. To capture '50.5 percent', a rule developer would prefer P: Percentage over other rules such as P: Number L: 'percent'. Capturing the percentage and date tokens with their respective pre-builts is more meaningful than capturing them with primitives expressing their syntax, e.g. token gap and regex. Moreover, developers build pre-builts to capture semantic concepts. Hence, non-semantic, generic primitives, i.e. token gap and regex, are ranked the lowest on the semantic scale. Literals and dictionaries capture the token's exact string and hence lie in the middle of the scale. Dictionaries are binned together with literals because they are essentially sets of literals. There are also rankings amongst the different types of concepts in the pre-built type. Pre-builts are ranked based on *generality* and *specificity*. Generic pre-builts such as P: CapitalWord are ranked lower than more specific ones such as P: Organization.

On the other hand, if a token has no inherent meaning, then rule developers least prefer pre-builts. The syntactic tokens 'in' and 'during' would most likely be described as token gaps by the rule developer, since (1) there is no pre-built describing the middle tokens, 'in' and 'during', and (2) the middle tokens are not attached to any semantics and play a syntactic role in the extractions. In general, generic, non-pre-built primitives are preferred for syntactic tokens.

A token that appears across multiple positive examples but not in any negative examples is also a semantic token even if a pre-built cannot capture it. In general, it is preferable to capture these tokens by more specialized primitives such as literals instead of generic ones such as token gaps; this



(a) The dashed lines point to candidate nodes for '8'.

(b) The final tree generated from the positive example, '8 percent'.

Figure 3. Tree generation from positive example, '8 percent'.

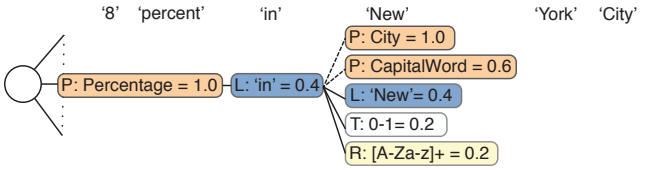


Figure 4. Tree generation for '8 percent in New York City' assuming the user provided a negative example, '50.5 percent in New Haven'. Dashed lines point to candidate nodes.

way, the commonality of the token across multiple examples is captured in a more precise way within the rule.

SEER learns primitives that can capture multiple consecutive tokens, e.g. P: Percentage captures the two consecutive tokens: '8 percent'. The score of the primitive is independent of the number of tokens it can capture.

SEER'S LEARNING ALGORITHM

At a high-level, the learning algorithm enumerates primitives per token for each of the positive examples, and combines them into sequences to form rules. Specifically, for each positive example, possible rules are heuristically generated and represented as tree structures. A *tree* is a representation of all the possible rules. An example tree generated from the positive example '8 percent' is shown in Figure 3(b). Each level of a tree corresponds to a token in the example, where all primitives in that level capture that token. Each path of ordered primitives from the root of the tree to one of its leaves encodes a rule. Once a tree is generated for each positive example, SEER *intersects* and refines the rules, which are then suggested to the user. Next, we discuss each step in detail.

Tree Generation

Given an example, a naive algorithm would generate a tree that encodes all possible rules capturing that example. For example, all primitives covering the first token of the positive example, i.e. '8', are added as the root's children (Figure 3(a)); each primitive capturing '8' points to all the possible primitives capturing the next token, 'percent'. SEER avoids the combinatorial blowup inherent in such a naive algorithm by performing optimizations at each level of the tree, in a careful manner that maintains the *diversity* and the *validity* of the generated rules. *Rule diversity* ensures that the suggested rules capture different sets of extractions. Specifically, a diverse set of rules should not contain rules composed of mainly one type of primitive; for instance, a non-diverse set of rules would only contain pre-built primitives. *Rule validity* ensures that the rule does not capture a negative example.

To maintain diversity, SEER keeps the highest scoring primitives per type when adding candidate children primitives to a node. To illustrate, suppose the algorithm is selecting primitives for the root node, i.e. token '8' in Figure 3(a). The highest scoring primitive amongst the pre-builts is P: Percentage, which is added to the tree, while the other pre-builts are discarded. R: [0-9]+ and L: '8' are kept in the tree, since there is only one primitive per type. For tokens that are not in the beginning or end of the example, a T: 0-1 is added (which doesn't apply to token '8' or 'percent').

To prevent capturing negative examples, SEER tests whether adding a candidate primitive node to a path will lead to creating a rule capturing a negative example. Intuitively, the likelihood of such a partial path, $R_{partial}$, capturing a negative example depends on the current position in the tree and which parts of the negative example can be captured by $R_{partial}$. Suppose the user wishes to extract percentages in New York City, and she provides e_{pos} = '8 percent in New York City' as a positive example and e_{neg} = '50.5 percent in New Haven' as a negative example. Suppose SEER is adding primitives to the token 'New' on the partial path P: Percentage L: 'in' (see Figure 4). Only one of the two pre-builts P: City and P: CapitalWord will be added, while the other primitive types will be added to the tree in order to maintain diversity. P: City is ranked highest by the default rankings, but picking it would create a rule that also captures e_{neg}.

Hence, the algorithm must check the following: (1) whether the beginning part of e_{neg} is captured by $R_{partial}$ and (2) whether $R_{partial}$ would capture e_{neg} despite what future primitives may be added after. The second condition specifically checks if the substrings in e_{pos} and e_{neg} that haven't been matched by $R_{partial}$ are equal, denoted as S_{neg} and S_{pos}, respectively. If $S_{neg} = S_{pos}$, then any rule formed after will capture both the positive and negative example.

Given $R_{partial}$ is P: Percentage L: 'in' P: City, we check the two conditions: 1) $R_{partial}$ captures the beginning part of e_{neg}; 2) $S_{neg} = S_{pos}$, since $R_{partial}$ captures e_{pos} and e_{neg}. Since all the conditions are satisfied, P: City is discarded from the tree, and the algorithm tests the same two conditions for the next highest scoring pre-built primitive, P: CapitalWord. While the pre-built passes the first condition, it fails the second condition, because S_{neg} ='Haven' and S_{pos} = 'York City'. Hence, P: CapitalWord is added instead of P: City. By adding P: CapitalWord instead of P: City, the algorithm creates more precise rules, whereas adding the latter pre-built would only create rules capturing the negative example.

To summarize, when SEER adds primitives as children to another primitive, it adds the highest scoring primitive per type such that the path from the root to that primitive doesn't capture negative examples partially (or fully). In this way, SEER does not enumerate all possible rules, while maintaining the diversity and validity of rules. The full algorithm is included in the supplemental material.

Tree Intersection

Once SEER generates trees representing the possible rules for each positive example, all trees are **intersected** with each

other, resulting in the creation of a new tree, the intersection tree (Algorithm 1). The intersection tree contains the rules capturing all the positive examples of the intersected trees, e.g. intersecting a tree capturing '8 percent' with a one capturing '50.5 percent' will result in a tree containing only the rules capturing both examples (Figure 5). The goal of the intersection operation is to group "similar" examples together, in order to both reduce the number of rules shown to the user, and to create more semantically meaningful rules.

Two trees are intersected by simultaneously traversing the trees in depth-first order and intersecting individual primitives (Algorithm 1). Primitives intersect if they are *identical*. In other words, primitives intersect with themselves, e.g. P: Percentage only intersects with itself. A dictionary can intersect with another one containing the exact same dictionary entry set, e.g. D: {up, grow} cannot intersect with D: {down}. Similarly, L: 'up' can only intersect with itself, and L: 'up' and L: 'down' do not intersect.

While the intersect operation combines identical primitives, the **merge** operation creates dictionaries from literals and dictionaries containing non-equal entries/string values e.g. merging L: 'up' and L: 'rise' results in D: {up, rise} or merging L: 'up' and D: {rise, grow} results in D: {up, rise, grow}. By allowing unequal literals and dictionaries to merge, SEER doesn't discard rules containing specific literals or dictionaries that only apply to one example, thus maintaining the diversity of the intersected rules, and moreover, it creates more semantically meaningful rules by generalizing literals into dictionaries. Moreover, without dictionary creation, rules containing mere consecutive exact strings matching exactly one example will be created but discarded during intersection. Note, user-supplied dictionaries are not merged, since they are considered as pre-builts.

If there are more than two trees to intersect, subsequent trees are intersected with the intersected tree (Algorithm 2). If the *intersection* of two trees does not contain any rules, SEER splits the tree's examples, resulting in a disjunction of examples and multiple intersected trees. Suppose e_1 = '8 percent up', e_2 = '50.5 percent up', and e_3 = 'increases 30 percent'. Since rules from e_1 and e_2 don't *intersect* (although they may *merge*) with any of the rules in e_3, the algorithm will return two **suggestion sets**, one for each positive example subset, where the first positive example subset is $\{e_1, e_2\}$ and the second subset is $\{e_3\}$.

Note that SEER disjuncts on null intersections even if merging the trees does not result in null. Intersecting trees of e_1 ='8 percent' and e_2 ='in 2008' results in an empty intersection tree, or two separate suggestion sets, S_{e_1} and S_{e_2}. While merging rules from e_1 and e_2 results in a suggestion set containing one rule, D: {8, in} D: {percent, 2008}, it is not as diverse as S_{e_1} and S_{e_2}. By performing a disjunction on null intersected trees, SEER avoids *over-merging* and creating non-diverse suggestion sets dominated by meaningless dictionary primitives.

Trimming Rules for the Final Suggestions

After intersecting the rules, there may be too many rules in each suggestion set to present to the user. For each suggestion set, its rules are trimmed, in such a way that maintains diversity. Trimming involves two steps: (1) grouping rules of the same primitive composition, a process called **classifying**, and (2) selecting the highest scoring rule from each group, where the *score of a rule* is the average of all its primitive scores. The average is simple and efficient, and it factors the primitive scores, which partially captures its complexity. The rule, R: [0-9]+ T: 0-2 L: 'percent', has a classification group of {Regex, Token Gap, Literal}. Rules that are composed of exactly these three primitive types will fall under the same classification group. Classifying rules from the intersected tree in Figure 5 will not trim any rules, since each rule belongs in its own classification group. Afterwards, the rules are ranked by their scores in descending order and presented to the user.

Algorithm 1: SEER's Intersection & Merge Algorithm

Input:
P_1, P_2: primitives from the input trees to intersect T_1 and T_2; The tree variables T_1 and T_2 point to the root primitives.
P_i: primitive in the resulting intersect tree T_i
Output:
P_i: a primitive node due to an intersection or null
Function Call:
Traverse(T_1, T_2, null)

Function *Traverse* (P_1, P_2, P_i)
  if (*IsTokenGap*(P_1) ***and not*** *IsTokenGap*(P_2)) ***or*** (***not*** *IsTokenGap*(P_1) ***and*** *IsTokenGap*(P_2)) **then**
    Let P_{TG} be the token gap and P be the non-token gap
    $P_i' := Primitive(P_{TG})$
    $P_i.children := P_i.children \cup \{P_i'\}$
    foreach $c \in P_{TG}.children$ **do**
      Traverse(c, P, P_i')
  else if *IsIntersects*(P_1, P_2) ***or*** *IsMergeable*(P_1, P_2) **then**
    $P_i' := IntersectMergePrimitives(P_1, P_2)$
    $P_i.children := P_i.children \cup \{P_i'\}$
    foreach $c_1, c_2 : c_1 \in P_1.children, c_2 \in P_2.children$ **do**
      Traverse(c_1, c_2, P_i')
  /* Eliminate non-intersectable paths */
  if $P_i.children = \emptyset$ ***and not***(*areBothLeaves*(P_1, P_2)) **then**
    Remove(P_i)
  return P_i

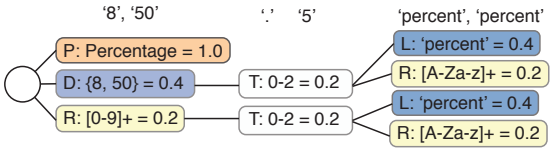


Figure 5. Intersected Tree for '8 percent' and '50.5 percent'.

Complexity Analysis

The complexity of SEER's learning algorithm depends on the number of positive examples, e, negative examples, n, and tokens per example, t. The tree generated for each positive example is of size at most 4^t. The tree has at most t levels for each token in the example and a maximum branching factor of four for each of the possible primitives: pre-builts, literals, token gaps and regular expressions. Each positive example generates a tree. The negative examples only limit the number of primitives at each node of the tree. Thus, negative exam-

Algorithm 2: Forming Suggestion Sets with Intersection

Definitions:
A suggestion set is $s = (P_s, R_s)$, where $P_s \subset P$, the set of all the positive examples, and R_s contains rules capturing all examples in P_s
Input:
T: set of trees to intersect
Output:
I: a list of suggestion sets

Function *Intersect(T)*
- $I := []$
- **foreach** $t \in T$, *t learned from example e* **do**
  - **if** $I = \emptyset$ **then** $I := I \cup \{(\{e\}, t)\}$
  - **else**
    - *IsAdded* := **false**
    - **foreach** $s = (P_s, T_i) \in I$ **do**
      - **if** *IsIntersectable*(T_i, t) **then**
        - $s := (P_s \cup \{e\}$, *Traverse*(T_i, t, null))
        - *IsAdded* := **true**
        - **break**
    - `/* Disjunct when t does not intersect */`
    - **if** ***not*** *IsAdded* **then** $I := I \cup \{(\{e\}, t)\}$
- **return** I

Algorithm 3: Refinement Computation

Definitions:
The *cover rules* of an extraction, x, are the rules that can capture x
Global Variables:
LIMIT: the max number of refinements to show
Input:
$S = (P_s, R_s)$: a suggestion set, where each rule r has its own set of extractions X_r
Output:
F, the set of refinements

Function *GetRefinements (S)*
- $F := []$
- $C := X_{r_1} \cup X_{r_2} \cup ... \cup X_{r_n}$ s.t. $r_i \in R_s, 1 \leq i \leq |R_s|$
- **foreach** $x_c \in C$ **do**
  - *AddRefinement* = **true**
  - **foreach** $x_f \in F$ **do**
    - **if** *CoverRules*(x_c) = *CoverRules*(x_f) **then**
      - *AddRefinement* = **false**
    - **break**
  - **if** *AddRefinement* **then** $F := F \cup \{x\}$
  - **if** $|F| >$ *LIMIT* **then break**
- **return** F

ples contribute $O(e4^t n)$ to the runtime. The intersection and merging algorithm also produces an intersected tree of size at most 4^t. Since each example-generated tree is intersected once with the intersected tree of a suggestion set and in the worst-case none of the example trees intersect leading to e possible suggestion sets, SEER's runtime complexity is $O(e^2 4^t + e4^t n)$.

While the worst-case runtime is exponential in nature, the goal of our heuristics is less on runtime optimization but more on suggesting a small set of rules that are diverse. While optimization can be explored, it is not within the scope of the paper, since most extractions are less than ten tokens. From experience with VINERy, users do not create long rules (e.g., with length greater than 10) because they are difficult to maintain. Instead, users build rules capturing lower-level sub-concepts, save them as pre-builts, and reuse them in rules for higher-level concepts.

FILTERING RULES WITH REFINEMENTS

Given the suggested rules, it may still be time-consuming for the user to analyze each one. Therefore, SEER further assists the user by analyzing the results of all suggested rules and selecting a small set of extractions that help disambiguate the rules (see Algorithm 3). SEER allows the user to filter rules by accepting or rejecting these extractions, which are called *refinements*. A refinement, x, is an extraction captured by a subset of the suggested rules, referred to as *covering rule set*, or *covering rules* in short. A user can accept or reject a refinement, and in turn its covering rules are kept or filtered (not shown to the user). If the user rejects '8 percent', then its covering rules will not be shown in the suggestion list.

Note that a rule may appear in multiple covering rule sets even if the sets are different from each other, e.g. it is possible that refinements x_1 and x_2 both contain the rule R_1 in their covering rule sets. Hence, *conflicting selections* may arise where the user rejects a refinement that is covered by a rule that was accepted under a different refinement (or vice versa). In order to prevent such a state, SEER *disables* selections for refinements that a user has not yet accepted or rejected. The selection for a refinement is disabled if **all** of its covering rules have been accepted or rejected under other refinements.

To illustrate, assume a suggestion set of rules R_1, R_2, R_3, and R_4, and let its refinements and their covering rules be x_1: $\{R_1, R_2\}$ and x_2: $\{R_2\}$. If the user rejects x_1, then the non-rejected rules are R_3 and R_4 and the rejected rules are R_1 and R_2. SEER disables x_2 if at least one of the following conditions is true:

1. If all its covering rules have been rejected by user selections, or if the intersection between the refinement's covering rules and the non-rejected rules is null. For x_2, its covering rules have been rejected under x_1 ($\{R_2\} \cap \{R_3, R_4\} = \{\}$).

2. If all of its covering rules are accepted based on user selections, or if the difference between the non-rejected rules and the refinement's covering rules is null. For x_2, none of its covering rules have been accepted ($\{R_3, R_4\} - \{R_2\} = \{\}$).

Since at least one of the conditions passed, x_2 is disabled. Hence, if the user rejects x_1, then the user selection will be disabled (grayed out) for x_2.

EVALUATION

We conducted a comparative user study of rule suggestion with SEER against direct rule specification with VINERy (Visual Integrated Development Environment for Information Extraction) [20], a commercially available drag-and-drop interface for specifying VAQL extraction rules. For each user, we observed completion-time, precision and recall for different extraction tasks completed on either SEER or VINERy. We hypothesize that, with the help of SEER's example-driven rule synthesis and refinement, users complete text extraction tasks more quickly with better accuracy and coverage than with only manual rule creation.

Participants and Method

We recruited 13 participants. These subjects are familiar with basic programming: they were undergraduate students who have taken or were taking an 'Introduction to Computer Science' course. This is to ensure that they could use VINERy and build regexes. These subjects have neither used VINERy nor SEER before.

We exposed each participant to both tools, allotting one-hour per tool. For each tool, we provided a 30-minute training session. We randomized the presentation of tools across subjects; 7 subjects used SEER first and 6 used VINERy first. The same pre-builts were available on SEER and VINERy.

In the beginning of each training session, the participant was presented with a 5 minute tutorial video. The tutorial video for VINERy explained how to create extractions rules and the tutorial video for SEER explained how to provide examples to SEER and how to interpret the rules suggested by SEER. After presenting the video, the user practiced completing text extraction tasks on a mock dataset. The mock tasks for VINERy introduced each of the basic primitives and how to create sequence rules out of the primitives. The mock tasks for SEER trained the users how to highlight examples, understand the suggested rules, use the refinements to filter rules and choose the correct one.

After the training session for a tool, the participant completed 3 tasks given in ascending task complexity, and we limited each task to 10 minutes. Task instructions included descriptions of sample extractions and extraction variations. The tasks were designed such that they were feasible for novice users to complete in 10 minutes. We randomized the choice of dataset for each feature across subjects in order to minimize learning effect. The two datasets were IBM press releases and FBI press releases.

Task 1: Users either extracted (a) all currency amounts from the IBM data set, or (b) all percentages from the FBI data set. The complexity of this *easy* task is a single extraction rule with one primitive.

Task 2: Users either extracted (a) all cash flow and OEM (Original Equipment Manufacturer) revenues, or (b) all percentage increases or decreases in offences. The complexity of this *moderate* task is a single extraction rule with multiple primitives.

Task 3: Users either extracted (a) all yearly quarter phrases, or (b) all population ranges. The complexity of this *hard* task is a disjunction of two rules with multiple primitives each.

Table 1 lists the expected extraction rules and sample extractions for each of the tasks above. The tasks are representative of common information extraction tasks, since users performed all typical rule development steps: find extractions in the dataset, test rules, and verify results.

After the participant finished the experiment, we asked them to finish a questionnaire, in order to understand their experience using the tools and evaluate our design choices including the preference ordering of primitives.

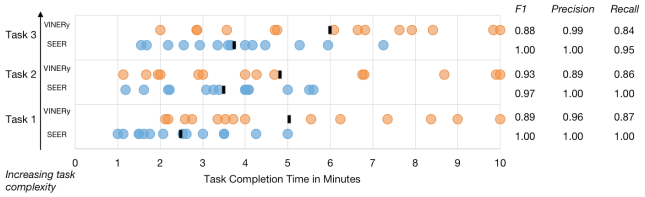


Figure 6. Time for task completion per participant. Black bars indicate mean times. The mean F_1, precision, and recall are also listed.

Results

We collected the duration of task completion and measured the participant's rules using the following standard metrics: $precision = \frac{t_p}{t_p+f_p}$ (also known as accuracy), $recall = \frac{t_p}{t_p+f_n}$ (also known as coverage), and $F_1 = 2 \cdot \frac{precision \cdot recall}{precision+recall}$, where t_p denotes the number of true positives, f_p denotes the number of false positives and f_n denotes the number of false negatives. See Figure 6.

We evaluate the results using the repeated measures ANOVA approach. Mauchly's test indicates that the preconditions (sphericity) for the ANOVA repeated measures have been met. We found a significant main effect of tool ($F_{1,41} = 19.452, p < 0.01$) for task completion times. For task completion times on Task 1 and Task 3, we found a significant main effect of tool ($F_{1,13} = 12.180, p = 0.004$; $F_{1,13} = 6.043, p = 0.029$), but no significant main effect of tool was found for Task 2. In Figure 6, notice that for all task complexity levels, the mean time to complete a task with SEER is lower than the mean time to complete the same task with VINERy.

We also performed a repeated-measures ANOVA of F_1 scores with tool as an independent factor and found a significant main effect of tool ($F_{1,40} = 9.367, p = 0.004$). For F_1 scores on Task 3, we found a significant main effect of tool ($F_{1,13} = 5.195, p = 0.04$), but no significant effects were found for Task 1 and Task 2. In Figure 6, we notice that for all task complexity levels, the mean F_1 scores for tasks completed with SEER are higher than the mean F_1 scores for the same tasks completed with VINERy.

User's programming experience (measured by asking how long a user has been programming) did not impact performance (duration and F_1 scores); no significant interaction effects were observed. In Figure 6, although it appears that a cluster of users spent a considerable amount of time in VINERy across task complexities, no correlations were found between the VINERy durations and programming experience or regular expression familiarity. The duration varied across complexity; for instance, some users took a long time on the first task and spent less time on the subsequent tasks.

We also recorded the number of times users utilized the refinement filters, and found that all users, except for one, used it regardless of task complexity. Users on average answered 2 refinement questions for easy tasks, 3 refinement questions for medium, and 4 for hard tasks, before converging to the desired rule. Only 1 participant did not use the refinements across all task complexities and went through 6 iterations in total (the most of any user) before selecting the desired rule,

#	Size	Complexity	Dataset	Expected Extraction Rule	Example Extractions
1	100	Easy	IBM	P: CurrencyAmount	$6 billion
1	100	Easy	FBI	P: Percentage	5.0 percent
2	30	Moderate	IBM	D: {OEM revenue, cash flow} T: 0-1 P: CurrencyAmount	cash flow was $4 billion
2	30	Moderate	FBI	L: 'offenses' T: 0-2 P: Percentage	offenses decreased 5 percent
3	30	Hard	IBM	D: {first, second, third, fourth} T: 0-1 L: 'quarter' T: 0-1 P: Integer OR	fourth-quarter 2005
				P: Integer D: {first, second, third, fourth} T: 0-1 L: 'quarter'	2007 first quarter
3	30	Hard	FBI	P: Integer L: 'to' P: Integer T: 0-1 L: 'population' OR	10 to 1,000 in population
				L: 'populations' T: 0-1 P: Integer T: 0-1 P: Integer	populations from 50 to 100

Table 1. Text Extraction Tasks. Size refers to the number extractions in the task.

when on average the other users did 4 iterations in total. The participant simply wanted to inspect each rule instead of filter by refinements.

Discussion

Users were most effective with SEER when they constructed simple extraction rules or complex disjunctions of multiple extraction rules. Users completed the simplest task more quickly in SEER than in VINERy, because in VINERy, most participants spent their time recreating sequence rules that did the equivalent of an existing pre-built they weren't aware of. Though after a few minutes of dragging and dropping primitives to create sequence rules, some of the participants noticed a pre-built that would complete the task. Hence, in Task 1 SEER had an impact duration-wise.

Users completed tasks requiring complex disjunctions of multiple rules significantly more quickly, accurately, and precisely with SEER than with VINERy. As the task required more variations, the participants took more time to create multiple rules that covered each variation and some even missed covering the details in the variations that SEER would have otherwise picked up on. For instance, in Task 3 of the IBM dataset, some of the target extractions have dashes in between the year's quarter and the literal 'quarter', e.g. 'Fourth-quarter'. Yet, some participants didn't include a token gap that would capture that variation. With SEER, participants simply made sure to highlight all the variations, which was easier and less error-prone than creating rules and simultaneously keeping track of those variations.

Qualitative Evaluation

After the participants finished all the tasks on both tools, we asked them to complete a questionnaire evaluating the experiences in both tools, asking them on their preference of primitives and for feedback on the interfaces.

SEER's Ranking Primitives

We hypothesized that users prefer some primitives over others depending on the example token type; this assumption drove the design of SEER's rule scoring. We asked the users, "If you can correctly extract the required text with all five primitives, pre-builts, dictionaries, literals, token gaps and regular expressions, which primitives would you prefer to use? Rank the primitives by your preference." By asking users to order primitives by their preference we affirmed our design choice. Figure 7 shows the ranking distributions and means.

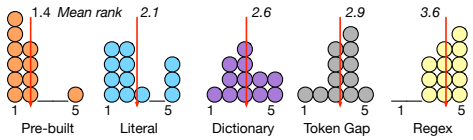


Figure 7. Distribution of participants' primitive rankings. The missing dots are due to missing survey responses.

Users ranked pre-builts the highest, literals the second highest, dictionaries the third, and regexes and token gaps the lowest, which matched closely with SEER's hypothesized default ranking of primitives. Similar to SEER's rankings, user's regex ranking contrasted the pre-built ranking, since pre-builts describe semantic features and regex's capture syntactic features devoid of meaning. One user said, "Regexes are the most expressive. Pre-builts are the fastest/most powerful." Users ranked literals after pre-builts, because, although not as powerful as pre-builts, it simulates searching and closely matches target extractions.

Thorough IE with SEER

SEER reinforces a more thorough approach to information extraction: (1) users inspected the dataset for actual instances of the target extractions and (2) users verified the extractions. In the following sections, we discuss these two aspects in detail.

Target Extraction Inspection. In the form of a Likert scale question from 1 (many examples) to 5 (few examples), we asked participants: (1) whether they felt they *ought to provide* many or few examples for SEER to suggest good rules and (2) whether they felt they *actually provided* too many or too few examples. In response to the two questions, users felt that they need not provide many examples ($\mu = 3.93$), but felt that they ended up providing more examples than necessary ($\mu = 3.43$). Users explained that the number of examples they provided largely "depend[s] on the variations". In Figure 8, the distribution of the total number of positive examples provided are shown. The majority of the users provided 3 to 4 positive examples per task.

Similarly, in VINERy, we asked users (1) how many examples (including actual instances in the dataset in addition to the sample extractions provided in the task instructions) were *necessary* to examine to construct a rule, and (2) how many examples they *felt* they actually examined to create a rule. With VINERy, unlike SEER, users did not think it was necessary to

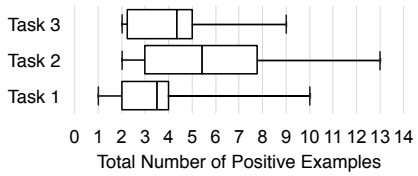


Figure 8. Distribution of the total number of positive examples.

inspect any actual extraction instances aside from the sample extractions given to them from the task description (μ = 3.36). Consequently, VINERy users felt that they did not inspect any dataset extraction instances or felt the need to search for extra variations from the dataset (μ = 3.38); one participant said, "I didn't scroll through the document for other possible examples. I just took the examples I was presented [in the task description]." Another said, "I never felt like I examined too many examples," and another said, "Just about right [number of examples]." The number of examples to create a rule was irrelevant to users because they knew instantly which primitives they wanted to build their rule, "Often, I knew straight away what I wanted, [for] example an integer number or date."

Extraction Verification. Participants had different methods of verifying extractions, ranging from a methodical search of known constant terms to mere glances at the extractions. One participant said, he checked "only the first few instances to see if the sequence satisfies my needs." Another one expressed a similar sentiment saying, "No, I checked for few [extractions] and I knew it would work." Others said they used the search functionality just to be sure, "I sometimes use the search functionality to make sure."

To evaluate extraction verification on both tools, we asked users how much they *felt* they verified and compared participants' answers with the *actual* percentage of extractions they analyzed. Specifically, we asked participants if they "searched through the whole dataset to make sure the extraction results contained all instances required by the text extraction task" on a Likert scale of 1 (strongly disagree) to 5 (strongly agree). The actual percentage of analyzed extractions was calculated by counting the number of extractions seen by the user.

We found that while the reported amount of extractions users *felt* they checked in VINERy (μ = 3.21) was more than in SEER (μ = 3.00), users in SEER *actually* checked, on average, 23% more extractions than users in VINERy. In both the easy and medium tasks, users were more likely to check their extractions in SEER than in VINERy (17% and 43% more actual extractions analyzed in easy and medium tasks, respectively). Yet, as the number of variations increased in the required task, users felt the need to verify their extractions in both tools; users in SEER and in VINERy checked, on average, 60% of the extractions. Note that while the number of extractions per task complexity differs, the number of extractions across task complexity were the same.

Participants reported that they inspected more extractions in VINERy than in SEER, yet the actual amount of extractions checked were less in VINERy than in SEER (except for the hard task). There was a tendency for VINERy users to *think* they analyzed the extractions. For instance, one VINERy user said, "I wanted to make sure that my rule was correct so I double checked a lot," yet his precision value did not reach 100 percent (96%). VINERy users had confidence in their rules, including incomplete ones, and hence, were more likely to *feel* as if they had actually inspected more extractions in VINERy than in SEER.

Creating Rules in VINERy

There are two major steps to a user's thought process during rule creation that SEER's learning algorithm emulates: (1) Sample target extractions are mentally tokenized and pieced together to create an *initial rule*, (2) Rules are edited in *iterations* to generalize across *variations*.

Initial Rule. In the questionnaire, we asked VINERy users on their thought process when creating rules. When working with VINERy, users mainly applied the same steps to creating an *initial rule*: (1) Mentally breakdown the examples into individual tokens, (2) Figure out which primitives to use, and (3) Combine the primitives into a pattern. One participant said, "[I] Look at examples, separate each word, and put it together," and another said, "I broke down an example into parts and added components."

When figuring out which primitive to use, users often picked the most meaningful primitive and the one closest to the target extraction. Users created a literal first in 49% of the tasks, a pre-built first in 38% of the tasks, and a dictionary first in 13% of the tasks. One user said, "Add literal, add dictionary, merge together." and another said, "I chose the parts that were easiest to use." Majority of the users clicked on the literal primitive first since it is only an exact string match and it was the easiest primitive to match to the target extraction.

Iterations and Variations. During rule construction, users think in iterations. One user said, "[I] identify patterns, create rules matching pattern, run rules, spot irregularities, debug." Another one said, "[I] look at target words, identify the constant words, identify variables, construct rule, try it out, check and modify." If the first run of the rule was not sufficient, users iteratively modified it until the rule was satisfactory. On the other hand, some users tried to create the rule that included all the variations on the first try. One user responded "[I] try to think everything through while creating the rules, because once they're created, changing [them] would be troublesome."

We also asked users to rate on a Likert scale from 1 (strongly disagree) to 5 (strongly agree) whether they deleted the rule instead of fixing the already-created rule when it did not capture an example from the documents . Again, we found users almost always iteratively fixed their rule ($\mu = 1.92$), but in some cases, "I usually tried to fix it, but if I hit a deadend, I delete all and start again."

RELATED WORK

Previous works on data extraction over text documents fall along a spectrum of methods: on one end, in the rule-based approach, developers manually create data extraction scripts or rules; on the other end, in Machine Learning (ML)-based approaches, systems learn complex mathematical models to extract the data. See Chang et al. [8] for a survey.

Recent research in IE has gravitated towards ML techniques. Example statistical models used for IE include Markov Logic Networks (MLN), Hidden Markov Models (HMM), Conditional Random Fields (CRF) [23, 22, 18]. However, since ML techniques need large training datasets and use complex, opaque models occasionally returning unexplainable results, industrial systems opt for models that employ simple, maintainable rules [9]. Rule-based systems depend on domain experts to define logical rules to capture possible variations in target extractions, which is a labor-intensive and time-consuming process. SEER synthesizes rules, taking advantage of the easy-to-understand model without requiring large training datasets. As such, SEER falls under the research agenda proposed in [9] of utilizing ML techniques for learning using a standard rule language as target model, in lieu of opaque statistical models.

Most prior work on IE rule synthesis is supervised [7, 10, 19], requiring the user to fully label a large set of documents. Whisk [28] is a semi-supervised rule synthesis system where only partially labeled data is required. Unlike Whisk, which works well only when hundreds of examples are provided, SEER works with only a small number of examples. Also related are systems for inducing regular expressions [25, 4, 5, 21]. Regular expression learners are limited to syntactic features of text; SEER capitalizes on semantic features of the text whenever possible and beneficial to data extraction.

A number of IE techniques (both rule-based and ML-based) take advantage of syntactic information, especially dependency grammars and trees [2, 13, 6, 11, 26]. Dependency trees contain part-of-speech tags and lexical values of the sentence's words. PropMiner [2] helps users unfamiliar with deep syntactic information to define rules consisting of SELECT, FROM, and WHERE clauses that query a sentence's dependency tree. PropMiner generates a very specific rule from a single example, which users must manually edit in order to generalize it to other target extractions, whereas SEER automatically outputs rules that generalize well. Extending SEER to consider rules that leverage syntactic parsing information is an interesting direction for future work.

In DeepDive [23], a knowledge base construction system based on Markov Logic Networks (MLNs), users write queries extracting entities and their relations. Such extractions are used to populate a data structure called a factor graph that facilitates statistical inference of other candidate extractions to be added to the knowledge base. MindTagger [27], a tool that was built on top of DeepDive, presents candidate extractions to the user in a way that allows for easy error analysis. In DeepDive and MindTagger, wrong extractions are tolerated as each one is weighted by confidence. SEER uses heuristics to remove rules that extract negative examples.

Heer et al. [12] describe a framework for data transformation called *Predictive Interaction* in which users iteratively highlight features of interest in the data transformation, and in turn the system suggests the next possible steps. Wrangler [15], which follows the Predictive Interaction framework, suggests relevant transformations from current user interactions on structured tabular data. In Wrangler, users can build a data cleaning script by directly interacting with the data, e.g. highlighting text, etc. SEER follows this idea closely; users highlight extraction texts and SEER suggests rules.

SEER's mixed-initiative interface, which combines example-driven synthesis and direct manipulation of extraction rules, is similar to Wrangler's and DataPlay's philosophy that applies mixed-initiative interfaces to other data problems: transformation and querying respectively [15, 1, 14]. SEER is also similar to other human-in-the-loop systems, such as ModelTracker [3], which visualizes ML models to allow developers to easily catch common ML mistakes (e.g. mislabeled data or feature deficiencies), and Kristjansson's [17] work, an interactive IE framework for form-filling where system detects low-confidence extractions for the user. While ModelTracker deals with ML models and Kristjansson's work focuses on assisted form-filling, SEER deals with the rule-based method and reduces the time to create and debug IE rules.

FlashExtract [19] is most similar to SEER. It allows users to highlight text they wish to extract and synthesizes a fixed data extraction script. SEER differs from FlashExtract as it only relies on the pattern of the extraction and not on the content surrounding the extraction (e.g. preceding ':' or on newline). This makes SEER work just as well for both structured documents, e.g. CSVs, logs, etc., and unstructured documents, e.g. reviews, press statements, etc. While some of the mentioned systems have extractors, including data transformation tools such as Wrangler and OpenRefine [24], they do not reveal how they synthesize extraction rules and like FlashExtract they expect semi-structured files like logs or CSVs.

CONCLUSION AND FUTURE WORKS

In this work, we presented SEER, a system that learns IE rules from a small set of user examples. We evaluated SEER against VINERy, a commercial tool that allows users to manually create rules in the same language as SEER and demonstrated that users completed extraction tasks in less time and with fewer errors with SEER. We demonstrated that the scoring and ranking of primitives in SEER's rule learning algorithm closely matches how rule developers select primitives when constructing rules. We combined rule learning with refinements to help users differentiate between rules and quickly select the appropriate one.

While this paper focused on an IE workflow centered around user-specified examples, SEER can improve on a number of aspects, including its synthesizer's efficiency: currently, SEER does not learn from previous iterations, i.e. when a user clicks on the learn button, it simply reruns the learning algorithm. We plan to explore how to efficiently maintain and exploit what was learned from previous iterations, as well as optimize the run-time performance of the learning algorithm to ensure interactivity even with examples containing several tokens. SEER is currently limited to learning sequences of primitives: we plan to extend its learning algorithm to learn rule predicates, other regular-expression quantifiers and modifiers.

Acknowledgments We acknowledge the generous support of the U.S. National Science Foundation under award IIS-1420941.

REFERENCES

1. Azza Abouzied, Joseph Hellerstein, and Avi Silberschatz. 2012. DataPlay: Interactive Tweaking and Example-driven Correction of Graphical Database Queries *(UIST '12)*. 207–218.
2. Alan Akbik, Oresti Konomi, and Michail Melnikov. 2013. Propminer: A Workflow for Interactive Information Extraction and Exploration using Dependency Trees. In *ACL*. ACL, 157–162.
3. Saleema Amershi, Max Chickering, Steven M. Drucker, Bongshin Lee, Patrice Simard, and Jina Suh. 2015. ModelTracker: Redesigning Performance Analysis Tools for Machine Learning *(CHI '15)*. 337–346.
4. Alberto Bartoli, Andrea De Lorenzo, Eric Medvet, and Fabiano Tarlao. 2016. Inference of Regular Expressions for Text Extraction from Examples. *IEEE Trans. on Knowl. and Data Eng.* 28, 5 (May 2016), 1217–1230.
5. Falk Brauer, Robert Rieger, Adrian Mocan, and Wojciech M. Barczynski. 2011. Enabling Information Extraction by Inference of Regular Expressions from Sample Entities *(CIKM '11)*. 1285–1294.
6. Razvan C. Bunescu and Raymond J. Mooney. 2005. A Shortest Path Dependency Kernel for Relation Extraction *(HLT '05)*. 724–731.
7. Mary Elaine Califf and Raymond J. Mooney. 2003. Bottom-up Relational Learning of Pattern Matching Rules for Information Extraction. *J. Mach. Learn. Res.* 4 (Dec. 2003), 177–210.
8. Chia-Hui Chang, Mohammed Kayed, Moheb Ramzy Girgis, and Khaled F. Shaalan. 2006. A Survey of Web Information Extraction Systems. *IEEE Trans. on Knowl. and Data Eng.* 18, 10 (Oct. 2006), 1411–1428.
9. Laura Chiticariu, Yunyao Li, and Frederick R. Reiss. 2013. Rule-Based Information Extraction is Dead! Long Live Rule-Based Information Extraction Systems! *(EMNLP 2013)*. 827–832.
10. Fabio Ciravegna. 2001. Adaptive Information Extraction from Text by Rule Induction and Generalisation *(IJCAI'01)*. 1251–1256.
11. Aron Culotta and Jeffrey Sorensen. 2004. Dependency Tree Kernels for Relation Extraction *(ACL '04)*. Article 423.
12. Jeffrey Heer, Joseph M. Hellerstein, and Sean Kandel. 2015. Predictive Interaction for Data Transformation. In *CIDR*.
13. Raphael Hoffmann, Luke S. Zettlemoyer, and Daniel S. Weld. 2015. Extreme Extraction: Only One Hour per Relation. *CoRR* abs/1506.06418 (2015).
14. Eric Horvitz. 1999. Uncertainty, action, and interaction: In pursuit of mixed-initiative computing. *IEEE Intelligent Systems* 14, 5 (1999), 17–20.
15. Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. 2011. Wrangler: Interactive Visual Specification of Data Transformation Scripts *(CHI '11)*. 3363–3372.
16. Rajasekar Krishnamurthy, Yunyao Li, Sriram Raghavan, Frederick Reiss, Shivakumar Vaithyanathan, and Huaiyu Zhu. 2009. SystemT: A System for Declarative Information Extraction. *SIGMOD Rec.* 37, 4 (March 2009), 7–13.
17. Trausti Kristjansson, Aron Culotta, Paul Viola, and Andrew McCallum. 2004. Interactive information extraction with constrained conditional random fields. In *AAAI*, Vol. 4. 412–418.
18. John D. Lafferty, Andrew McCallum, and Fernando C. N. Pereira. 2001. Conditional Random Fields: Probabilistic Models for Segmenting and Labeling Sequence Data *(ICML '01)*. 282–289.
19. Vu Le and Sumit Gulwani. 2014. FlashExtract: A Framework for Data Extraction by Examples *(PLDI '14)*. 542–553.
20. Yunyao Li, Elmer Kim, Marc A. Touchette, Ramiya Venkatachalam, and Hao Wang. 2015. VINERy: A Visual IDE for Information Extraction. *Proc. VLDB Endow.* 8, 12 (Aug. 2015), 1948–1951.
21. Yunyao Li, Rajasekar Krishnamurthy, Sriram Raghavan, Shivakumar Vaithyanathan, and H. V. Jagadish. 2008. Regular Expression Learning for Information Extraction *(EMNLP '08)*. 21–30.
22. Andrew McCallum, Dayne Freitag, and Fernando C. N. Pereira. 2000. Maximum Entropy Markov Models for Information Extraction and Segmentation *(ICML '00)*. 591–598.
23. Feng Niu, Ce Zhang, Christopher Ré, and Jude W Shavlik. 2012. DeepDive: Web-scale Knowledge-base Construction using Statistical Learning and Inference. *VLDS* 12 (2012), 25–28.
24. OpenRefine 2010. OpenRefine. (2010). http://openrefine.org/.
25. Paul Prasse, Christoph Sawade, Niels Landwehr, and Tobias Scheffer. 2015. Learning to Identify Concise Regular Expressions That Describe Email Campaigns. *J. Mach. Learn. Res.* 16, 1 (Jan. 2015), 3687–3720.
26. Alexander Schutz and Paul Buitelaar. 2005. RelExt: A Tool for Relation Extraction from Text in Ontology Extension *(ISWC'05)*. 593–606.
27. Jaeho Shin, Christopher Ré, and Michael Cafarella. 2015. Mindtagger: A Demonstration of Data Labeling in Knowledge Base Construction. *Proc. VLDB Endow.* 8, 12 (Aug. 2015), 1920–1923.
28. Stephen Soderland. 1999. Learning Information Extraction Rules for Semi-Structured and Free Text. *Mach. Learn.* 34, 1-3 (Feb. 1999), 233–272.