Supplementary Material for 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

Algorithm 1: Seer's Learning Algorithm

Terminologies : 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
Inputs : E_p: Set of positive examples; E_n: Set of negative examples
Output : S, a list of suggestion sets, s.t. each P_s is unique and the union of all $P_s = E_p$ (covers all positive examples)

Function *Learn (E_p, E_n)*
    Let T be a list of trees
    T = []
    foreach *e* ***in*** E_p **do**
        `/* Generate tree for example` e `*/`
        $T_e := GenerateTree(e)$
        $T.add(T_e)$
    `/* Groups and disjuncts rules and then trim */`
    Let S be a list of suggestion sets
    $S := Trim(Intersect(T))$
    return S

Algorithm 2: Tree Generation for an Example

Global Variables : E_n: all the negative examples
Input : e, the example the tree generation will learn from
Output : T, the generated tree

Function *GenerateTree (e)*
    Let K_e be a list of tokens
    $K_e = Tokenize(e)$
    Let T be a tree and C_T be a pointer to nodes in T
    $C_T := T.root$
    Let R be a list of primitives from the root to C_T
    $T = GrowTree(C_T, K_e[0], R, e)$
    return T

Function *GrowTree (C_T, k, R, e)*
    $P_k = LearnPrimitives(k, R, e)$
    $C_T.children := C_T.children \cup P_k$
    foreach *c* ***in*** $C_T.children$ **do**
        $R.append(c.primitive)$
        if ***not*** *IsLastToken(t)* ***or not*** *IsCaptureNegativeExample(R)* **then**
            Let P_k be a primitive
            $P_k := GrowTree(c, r.nextToken(), R, e)$
        $R.remove(c.primitive)$
        if $P_k =$ ***null*** **then** $C_T.remove(c)$
    if ***not*** *IsLastToken(t)* ***and*** $C_T.children \neq \emptyset$ **then return null**
    return $C_T.primitive$

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

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

Algorithm 3: Learn Primitives

Global Variables : E_n: all the negative examples, $e_n \in E_n$
Terminologies : $|e|$ denotes the the number of tokens in e; $e[i, j]$ denotes the ordered list of the i'th to j'th tokens from e, $1 \leq i, j \leq |e|$
Inputs : k is the token to learn from; R is the current path from the root to some primitive in the tree being generated; e is the positive example k is in
Output : P_k, set of primitives capturing k given the current path R

Function *LearnPrimitives (k, R, e)*
    $P_k := []$
    `/* Maintain diversity: iterate over all types (pre-built, dictionary, etc) */`
    foreach *t* ***in*** *all possible primitive types* **do**
        Let A_k be the set of primitives that capture k of type t
        Let P'_k be the highest scoring primitive from A_k s.t. $CapturesNegative(R,\ p_{A_k} \in A_k,\ e)$ is **false**
        $P_k := P_k \cup P'_k$
    return P_k

Inputs : R is the current path from the root to some primitive in the tree being generated; $p_{A_k} \in A_k$; e: the positive example the tree generation is learning from
Output : Returns **true** if appending p_{A_k} to the R will lead to creating rules capturing a negative example

Function *CapturesNegative (R, p_{A_k}, e)*
    $R' := R.append(p_{A_k})$
    Let $e_p[1, m]$ be the tokens R' captures
    if *IsRuleCaptures*$(e_n[1, k], R')$ ***and*** $e_n[k, |e_n|] = e_p[m, |e_p|]$ **then**
        `/* The rule that is forming will capture` e_n `*/`
        return true
    return false

Algorithm 4: 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**
                    $s := (P_s \cup \{e\}, Traverse(T_i, t, \text{null}))$
                    IsAdded := **true**
                    break
            `/* Disjunct when` t `does not intersect */`
            if ***not*** *IsAdded* **then** $I := I \cup \{(\{e\}, t)\}$
    return I

Algorithm 5: 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

Algorithm 6: Trimming From Suggestion Sets

Input : S: list of suggestion sets
Output : S, with less rules
Function *Trim(S)*
  foreach $s := (P_s, R_s) \in S$ **do**
    $G := Classify(R_s)$
    $R_{final} := []$
    foreach $g \in G$ **do**
      `/* g is a list of rules */`
      Let r be the highest scoring rule from g
      $R_{final} := R_{final} \cup \{r\}$
    `/* Rank rules by their scores in desc. order */`
    $Sort(R_{final})$
    $R_s := R_{final}$
  return S
Function *Classify*(R_s)
  Let G be a list of grouped rules
  $G := []$
  foreach $r \in R_s$ **do**
    IsAdded := **false**
    foreach $G_i \in G$ **do**
      `/* The classification group of a rule is the set of composing primitives of that rule */`
      if *ClassificationGroup*(G_i) = *ClassificationGroup*(r) **then**
        $G_i := G_i \cup \{r\}$
        IsAdded := **true**
        break
    if ***not*** *IsAdded* **then**
      $G_i' := \{r\}$
      $G := G \cup \{G_i'\}$
  return G

Algorithm 7: Refinement Computation

Terminologies : The *cover rules* of an extraction, x, are the rules that can capture x
Global Variables : *LIMIT*: the max num of refinements to show
Inputs : $S = (P_s, R_s)$: a suggestion set, where each rule r has its own set of extractions X_r
Output : F, the set of refinements
Function *GetRefinements (S)*
  $F := []$
  $C := X_{r_1} \cup X_{r_2} \cup \ldots \cup X_{r_n}, r_i \in R_s, 1 \le i \le |R_s|$
  foreach $x \in C$ **do**
    if *IsAdd(x, F)* **then** $F := F \cup \{x\}$
    if $|F| >$ *LIMIT* **then** **break**
  return F
Function *IsAdd (x, F)*
  foreach $x_f \in F$ **do**
    if $CoverRules(x_f) = CoverRules(x)$ **then** **return false**
  return true

Algorithm 8: Disable Refinements

Inputs : s is a suggestion set, where $s = (P_s, R_s)$; F is the list of all refinements
Function *DisableRefinement (s, F)*
  `/* Get the rules the user has rejected */`
  $R_{rejected} := GetRejectedRules(s.R_s)$
  $R_{display} := Difference(s.R_s, R_{rejected})$
  `/* For each refinement that the user hasn't accepted or rejected, disable appropriate refinements */`
  foreach $f \in F$ **do**
    if ***not*** *IsUserAcceptOrReject*(f) **then**
      $R_f := CoveringRules(f)$
      `/* If all the covering rules cannot be displayed, then disable. */`
      $isAllNotDisplayed := isEmpty(intersection(R_f, R_{display}))$
      `/* If all the covering rules can be displayed, then disable. */`
      $isAllDisplayed := isEmpty(difference(R_{display}, R_f))$
      if *isAllNotDisplayed* ***or*** *isAllDisplayed* **then**
        Disable(f)
      else
        Enable(f)