Appendices

A Features for Semantic Labeling

Below are the description of features used by SERENE semantic model. Note that features are extracted from the name and values of each attribute in the data source. Thus, all the data from a single column is reduced to some feature vector of the concatenation of values computed by the extractors below. We refer to a single value from the column as an entry. For example, in the table below, John is an entry of the column Person.

Table 2: Example table.

Person	Age	Gender
John	20	M
Jane	30	F

Table 3: Feature extractors that produce a scalar value

Feature name	Description
num-unique-vals	A simple count of unique entries of a column
prop-unique-vals	The proportion of unique entries of a column.
prop-missing-vals	The proportion of all missing/empty entries from all entries in a column.
ratio-alpha-chars	The average of the proportion of alphabetic characters from each entry.
prop-numerical-chars	The average of the proportion of numeric characters from each entry.
prop-whitespace-chars	The average of the proportion of whitespace characters from each entry.
prop-entries-with-at-sign	The proportion of entries with an '@' sign.
prop-entries-with-hyphen	The proportion of entries with a '-'.
prop-range-format	The proportion of entries which follow some numerical range format ([0-9]+)-([0-9]+)
is-discrete	A binary indicator if the column entries are discrete.
entropy-for-discrete-values	The entropy of the entries of a column.
shannon-entropy	Shannon's entropy for the vocabulary specified in the description of the feature 'char-dist-features'

Table 4: Feature extractors that produce a vector value. Some of these are computed together to reduce computation time. Others are values that are computed for all classes.

Feature name	Description
inferred-data-type	Produces a vector which contains the inferred data type. Only a single element is non-zero in the vector (1-of-K format). The set of inferred types are in float, integer, long, bool, date, time, datetime, string.
char-dist-features	Produces a vector for character distribution of column values; considered characters are printable characters
stats-of-text-length	Computes the mean, median, mode, min, max of string lengths from a columns entries.
stats-of-numerical-type	Computes the mean, median, mode, min, max of entries from a column of numerical type.
prop-instances-per-class-in-knearestneighbours	This finds the k columns from the training set with most similar names. It then produces a vector of class proportions from these k neighbours. Parameters: "num-neighbours": The number of neighbours to find (k).
mean-character-cosine-similarity-from-class-examples	For some instance t, computes the character distribution from this instance's entries, and produces a vector of the average cosine similarity of t with training instances of each class.
min-editdistance-from-class-examples	For some instance t, produces a vector of the minimum column name edit distance between t and all training instances of each class. Parameters: "cache-size": size of cache to store edit distances of words; "max-comparisons-per-class" : max number of training instances to compare for each class.
min-wordnet-jcn-distance-from-class-examples	For some instance t, produces a vector of the minimum column name JCN WS4J distance between t and all training instances of each class. Parameters: "cache-size": size of cache to store wordnet distances of words; "max-comparisons-per-class": max number of training instances to compare against for each class.
min-wordnet-lin-distance-from-class-examples	For some instance t, produces a vector of the minimum column name LIN WS4J distance between t and all training instances of each class. Parameters: "cache-size": size of cache to store wordnet distances of words; "max-comparisons-per-class": max number of training instances to compare against for each class.

B MiniZinc Model

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%                                                          %%
%% MiniZinc model to solve the Rel2Onto problem             %%
%%  as a composition of Steiner Tree Problem                %%
%%  and a matching problem                                  %%
%%                                                          %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

include "globals.mzn";

%%%%%%% Steiner Tree Problem
int: nbV;                               % Nb of nodes in alignment graph
int: nbE;                               % Nb of edges in alignment graph
set of int: nodes = 1..nbV;
set of int: edges = 1..nbE;
set of int: cnodes;                     % Class nodes in alignment graph
set of int: dnodes;                     % Data nodes in alignment graph
array [nodes] of var bool: vs;          % Is a node of the alignment graph in the solution?
array [edges] of var bool: es;          % Is an edge of the alignment graph in the solution?
array [nodes,edges] of bool: adjacent;  % adjacent[n,e] <=> node n is an endnode of edge e
array [edges] of nodes: heads;          % Endpoint 1 of each edge (the graph is undirected)
array [edges] of nodes: tails;          % Endpoint 2 of each edge
array[1..2,edges] of int: endpairs = array2d(1..2,edges,tails++heads);  % Endpoints of each edge
array [edges] of int: ws;               % Weights of edges in alignment graph
var 0..sum(ws): w;                      % Weight of the solution corresponding to the alignment part
constraint steiner_tree(vs,es,adjacent,endpairs,w,ws);  % Steiner Tree constraint
constraint forall (d in dnodes)
                   (vs[d] <-> sum(e in edges where adjacent[d,e])(es[e]) == 1);% Only 1 edges per data node

%%%%%%% Matching Problem
int: nbA;                                           % Nb of attribute nodes
set of int: atts = 1..nbA;
array [atts] of set of dnodes: attribute_domains;   % Data nodes to which each attribute can attach (usually all)
array [atts] of var dnodes: match;                  % Match between Attributes and Data nodes
array[atts,dnodes] of int: match_costs;             % Weight of match edges
constraint forall (a in atts) (match[a] in (attribute_domains[a]));   % Must match someone in the domain
constraint forall (a in atts) (vs[match[a]]);       % The data node matched must be in the subgraph of the alignment
constraint alldifferent(match);                     % Exact match
var 0..sum(a in atts)(sum(n in dnodes)(match_costs[a,n])): wm;  % Total cost from matching part :
constraint wm = sum(a in atts) (match_costs[a,match[a]]);
```

Model 1: MiniZinc model (part 1 of 2)

```
%%%%%%% Unknown nodes
int: nb_unknown_nodes;                  % Number of unknown data nodes
set of int: unknowns = 1..nb_unknown_nodes;
array [int] of int: all_unk_nodes;      % List of all unknown nodes (first two elements are class nodes, by convention)
int: node_root = all_unk_nodes[1];      % "Root" class node
int: node_unk = all_unk_nodes[2];       % "Unknown" class node
constraint if nb_unknown_nodes > 0 then
    vs[node_unk] <-> vs[node_root]      % If class node "Unknown" is used, class node "Root" is used as well
 /\ vs[node_root] <-> sum(e in edges where adjacent[node_root,e])(es[e]) in {1,2}
else true %Ignore constraint
endif;


%%%%%%% Patterns
int: nb_patterns;                       % Number of patterns
set of int: PATTS = 1..nb_patterns;
array[PATTS] of int: patterns_w;        % Support of patterns
array[PATTS] of set of int: patterns;   % The actual patterns
array[PATTS] of var bool: c_p;          % Is pattern chosen to be part of the solution?
constraint forall (p in PATTS) ((sum (e in patterns[p])    % All edges of a pattern are used <-> pattern is used
                                 (es[e]) == card(patterns[p])) <-> c_p[p]);
var 0..sum(patterns_w): wp;             % Total prize collected from patterns :
constraint if nb_patterns = 0 then wp = 0 else wp = sum(p in PATTS) (c_p[p] * patterns_w[p]) endif;


%%%%%%% Final objective function
var int: obj = w + wm - wp;             % Total = (cost of semantic model) - (prize collected from patterns)


%%%%%%% Search startegy
array[atts,dnodes] of int: match_costs_sorted;  % Match of attributes, sorted by cost.
ann: cheap_matches = int_search([ match[a] == match_costs_sorted[a,i] |a in atts, i in dnodes],
                                input_order,indomain_max,complete);

array[edges] of int: sorted_edges = arg_sort(ws);
ann: cheap_edges = int_search([es[sorted_edges[nbE + 1 - e]] | e in edges],input_order,indomain_min,complete);

ann: cheap_edges_fill_pattern;          % Custom search strategy for patterns (implemented in Chuffed)
ann: custom(ann, array[int] of int, array[int] of var bool);
array[int] of ann: cefp =
if nb_patterns == 0 then []             % This builds a list in a format understood by Chuffed, to work on the "es" vars
else [custom(cheap_edges_fill_pattern,[nbE]++[nb_patterns]++ws++[ card(patterns[p]) | p in PATTS]++patterns_w++array1d([
      e | p in PATTS, e in patterns[p]]),es)] endif;

% The solver will use these strategies in order:
% 1) Try using the cheapest matches, until a complete match is found (i.e. for all attributes)
% 2) If there are patterns, use a specific search called "cheap_edges_fill_pattern". (Implemented in Chuffed only)
%    It will start adding edges in increasing order of cost, while trying to fill patterns that are "worth" filling.
%    E.g. We already set es[x] = true for x in {e1,e2,e3}; if adding ws[e5] = 4 and ws[e7] = 6, but
%    {e1,e2,e3,e7} form a pattern p of patterns_w[p] = 3, then ws[e7] - patterns_w[p] <  ws[e5], so e7 is selected first
%    If there are no patterns, this strategy is ignored.
% 3) Try to eliminate edges by decreasing cost: it will eliminate the more expensive edges first, until
%    the strictly necesary cheap edges are added by the propagators. This is used only when there are no
%    patterns, otherwise (2) already selects the edges.

solve ::seq_search([cheap_matches]++cefp++[cheap_edges])
minimize obj;
```

Model 2: MINIZINC model (part 2 of 2)