Progress Report: Simplifying a Noisy schema.org Knowledge Graph

Sipan BAREYAN

sipan.bareyan@universite-paris-saclay.fr

Abstract

This progress report summarizes what has been built so far to shrink and clean a large noisy schema.org knowledge graph from Web Data Commons. Two working parts: (1) trim rarely useful types using a simple scoring loop over a type interaction graph; (2) score and filter predicates per remaining type with a small set of statistics plus a light neural helper. The current pipeline keeps central structure (e.g. Product, Offer, Book) while dropping long-tail clutter. Remaining work: better evaluation, key discovery, and safer entity cleanup.

1 Introduction

Public web RDF is messy: many types appear only a few times, predicates are inconsistently filled, and storage plus querying cost grows fast. The goal here is practical: reduce the graph while keeping information people actually use. Work to date:

- Count and connect types at an abstract level.
- Iteratively remove weak types.
- Score predicates for each type and keep the ones that add clear value.

Preprocessing was minimal: drop empty literals and entities with no type. Everything else builds on that baseline.

2 Methods

2.1 Type-Level Simplification

We build a compact graph: each node is a schema.org class (plus a Literal sink). Each original triple only contributes a count from the subject's type to the object's type (or Literal). This gives a small "type interaction" view instead of millions of entities.

For a triple (s, p, o) with $s : T_s$ and optional $o : T_o$ we increment an edge $T_s \xrightarrow{p} T_o$ or $T_s \xrightarrow{p} \texttt{Literal}$. We then normalize outgoing edges and (separately) the incoming perspective:

$$P_{\text{out}}(u \xrightarrow{p} v) = \frac{w(u,p,v)}{\sum w(u,p',v')}, \quad P_{\text{in}}(u \xleftarrow{p} v) = \frac{w(v,p,u)}{\sum w(v,p',u)}.$$

We use the PageRank algorithm to score the nodes. Two random walk passes (forward and reverse) produce prominence scores plus how often each (type, predicate, target) was traversed. For each type t we also keep:

- C_t: how many entities use the type.
- D_t: a depth weight from the chosen root (closer = higher). The formula is

$$D_t = \frac{1}{1 + \text{distance from the root}}$$

We combine them:

$$S_t = \left(\frac{C_t}{\sum_u C_u}\right)^{1/4} \cdot D_t^{1/2} \cdot (3\,PR^{\rightarrow}(t) + PR^{\leftarrow}(t))$$

Heuristics (roots, exponents, the 3:1 weight) were picked for stability, not tuned. After softmax we keep only the smallest prefix of types whose cumulative mass reaches a schedule fraction each round. Removing low contributors and rescoring sharpens the set. If an entity keeps multiple surviving types we pick the top as primary and move the rest to `additionalType` so later steps are not biased.

Algorithm 1 Multi-Round Type Refinement

```
1: Input: root type $R$, levels $L$
2: Build graph, restrict to nodes reachable from $R$
3: for $i = 1..L$ do
4:     Recompute edge probabilities; run forward & reverse walks
5:     Compute and softmax $S_t \rightarrow P_t$
6:     Keep minimal prefix with cumulative $\geq i/(L+1)$; remove rest
7: end for
8: Output surviving types
```

2.2 Predicate-Level Analysis

With the type list settled, we treat each type independently. For a type T and its predicates p we compute:

$$f_p = \text{fraction of entities with } p, \quad u_p = \text{distinct object values/total pairs}$$

$$h_p = \text{entropy of object value counts}, \quad q_p = \text{average fraction of other predicates present on entities using } p$$

$$r_p = \text{share of random walk traversals touching } (T, p, *)$$

We min–max scale h_p and q_p within the type (others kept raw). Then:

$$\text{structural}_p = f_p^{1/2}\, q_p\, \ln(1 + r_p), \qquad \text{data}_p = h_p u_p, \qquad \text{raw}_p = \text{structural}_p \cdot \text{data}_p$$

A temperature softmax with $\tau = 1/|P_T|$ gives weights w_p. We flag top predicates until a target cumulative weight Θ (e.g. 0.60) as "to be kept using score system".

A small neural model has been trained on a subset of decisions made using the method above, validated by manual review. This provides an additional mechanism to flag predicates for retention. The model takes as input f_p, u_p, h_p, q_p, r_p and has 2 output nodes representing the probabilities to keep or discard the predicate.

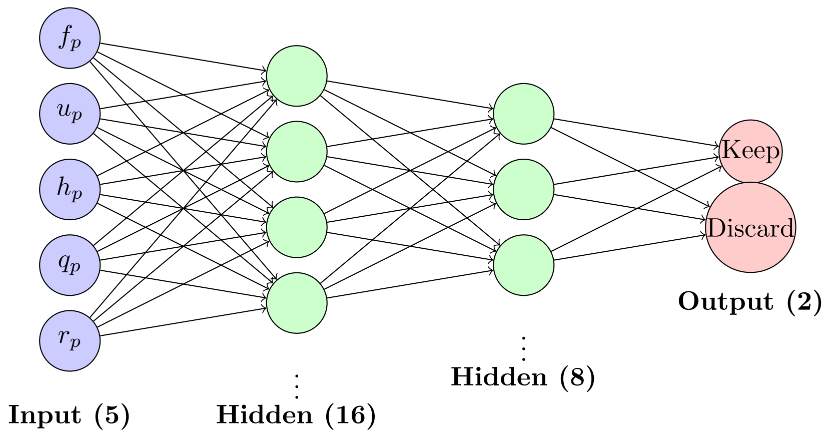

Figure 1: Neural network architecture for predicate filtering. The model takes 5 statistical features as input and outputs probabilities for keeping or discarding each predicate.

The neural model has 2 outputs, we denote the keep confidence score as p_p^{NN} . If model and score agree we accept. If the model wants to keep but the score rejects, we keep (bias toward recall). Otherwise we compute:

$$s_p = p_p^{\text{NN}}\left(1 + \frac{w_p}{\bar{w}_{\text{kept}}}\right)$$

and rescue if $s_p \geq 0.5$.

Algorithm 2 Hybrid Predicate Filtering

```
1: Compute $w_p$, identify score-kept set $K$, mean $\bar{w}_K$
2: for each predicate $p$ do
3:     Get $p_p^{\text{NN}}$
4:     if agreement then apply decision
5:     else if NN keeps then keep
6:     elsecompute $s_p$; keep if $s_p \geq 0.5$
7:     end if
8: end for
```

3 Results and Discussion

3.1 Type Pruning

The process reduced a large noisy class list to a smaller core set (e.g. Book, Product, Offer, Person, Organization). High-count but low-interaction tails were removed. Table 1 shows a slice of the final round applied to the "Book" class-specific dataset.

3.2 Predicate Filtering

Per-type scoring plus the neural helper kept core descriptive fields (e.g. Product: name, sku, brand; PostalAddress: streetAddress, postalCode, addressLocality) and dropped rarely used or low-impact ones. This adapts to the actual dataset instead of a fixed manual whitelist. Example tables are in the Appendix.

3.3 Suggested Next Steps

- Correlate predicates: co-use matrix to find strong pairs/groups.
- Flag and delete weak entities using a scoring mechanism based on the scores computed for each predicate for a specific type.

4 Implementation

The core logic is implemented in the rust-kg-explorer project. The system can load datasets from Web Data Commons and perform initial cleanup using the routines interface. Analysis begins from the analysis page, where users can visualize the class relations graph and apply the proposed algorithms for type analysis to retain only important types. The system also performs predicate analysis and filters out statistically insignificant predicates.

The source code is available at `https://github.com/bareyan/rust-kg-explorer`.

5 Limitations

- Heuristic weights and exponents not tuned.
- No semantic checks: statistically strong but semantically odd predicates may pass.
- Key inference and safe merging not yet implemented.

6 Conclusion

Current stage: a working, fast pipeline that shrinks type and predicate space while keeping central descriptive power. Next work: better evaluation, predicate grouping, key discovery, cautious entity merging.

A Result tables

Table 1: Results of the final round of type-level pruning. Core structural types are successfully identified and retained.

Type	**Count**	**Fwd PR**	**Rev PR**	**Decision**
`s:Book`	303,623	0.177	0.316	Kept
`s:ListItem`	259,971	0.219	0.052	Kept
`s:Person`	216,738	0.188	0.021	Kept
`s:Offer`	158,417	0.097	0.053	Kept
`s:Organization`	132,347	0.098	0.013	Kept
`s:WebPage`	48,172	0.032	0.164	Kept
`s:BreadcrumbList`	82,123	0.044	0.052	Kept
`s:AggregateRating`	59,733	0.042	0.006	Kept
`s:Product`	42,630	0.021	0.065	Kept
`s:EntryPoint`	59,089	0.046	0.003	Kept
`s:Website`	40,363	0.036	0.004	Kept
`s:CreativeWork`	15,303	0.008	0.111	Kept
`s:Comment`	29,620	0.035	0.002	Kept
`s:PostalAddress`	37,882	0.029	0.002	Kept
`s:Review`	31,300	0.020	0.021	Kept
`s:Country`	491	0.075	0.000	Kept
`s:Answer`	29,810	0.017	0.035	Kept
`s:PaymentMethod`	16,914	0.017	0.001	Discarded
`s:QuantitativeValue`	17,545	0.012	0.001	Discarded
`s:Article`	11,459	0.005	0.055	Discarded
`s:CollectionPage`	3,516	0.001	0.021	Discarded
...	...	...	...	Discarded

Table 2: Predicate analysis for the `s:PostalAddress` type. Note: Predicate URIs are shortened to `s:predicateName`.

Predicate	**Frequency**	**Uniqueness**	**Entropy**	**Score**	**Decision**
`s:streetAddress`	0.903	0.019	0.704	12.506	Kept
`s:name`	0.082	0.634	1.000	12.504	Kept
`s:postalCode`	0.885	0.015	0.673	12.503	Kept
`s:addressLocality`	0.905	0.016	0.619	12.501	Kept
`s:addressRegion`	0.701	0.008	0.246	12.497	Kept
`s:addressCountry`	0.261	0.013	0.411	12.496	Kept
`s:telephone`	0.012	0.009	0.101	12.496	Discarded
`s:email`	0.010	0.008	0.061	12.496	Discarded
`s:postOfficeBoxNumber`	0.001	0.120	0.020	0.000	Discarded
`s:description`	0.000	0.500	0.000	0.000	Discarded

Table 3: Predicate analysis for the `s:Product` type. The system correctly identifies core product attributes.

Predicate	**Frequency**	**Uniqueness**	**Entropy**	**Score**	**Decision**
`s:name`	0.999	0.893	1.000	6.512	Kept
`s:description`	0.820	0.865	0.967	6.357	Kept
`s:sku`	0.770	0.946	0.978	6.309	Kept
`s:brand`	0.530	0.855	0.900	6.285	Kept
`s:gtin13`	0.433	0.978	0.934	6.268	Kept
`s:url`	0.604	0.858	0.936	6.248	Kept
`s:mpn`	0.175	0.906	0.805	6.206	Kept
`s:productid`	0.113	0.992	0.790	6.202	Kept
`s:releaseDate`	0.070	0.743	0.698	6.202	Kept
`s:category`	0.111	0.213	0.436	6.201	Kept
`s:mainitemBarcode`	0.034	0.984	0.665	6.201	Discarded
`s:isbn`	0.023	0.835	0.554	6.201	Discarded
...	...	...	...	...	Discarded