Neural-Guided Program Synthesis of Information Extraction Rules Using Self-Supervision

Enrique Noriega-Atala[♠], Robert Vacareanu[♠, ♣], Gustave Hahn-Powell[♠], and Marco Antonio Valenzuela-Escárcega

[♠]University of Arizona, Tucson, AZ, USA
[♣]Technical University of Cluj-Napoca, Cluj-Napoca, Romania
{enoriega,rvacareanu,hahnpowell}@arizona.edu
macrovalenzuelaescarcega@gmail.com

Abstract

In this work we propose a neural-based approach for rule synthesis designed to help bridge the gap between the interpretability, precision and maintainability exhibited by rule-based information extraction systems with the scalability and convenience of statistical information extraction systems. This is achieved by avoiding placing the burden of learning another specialized language on domain experts and instead asking them to provide a small set of examples in the form of highlighted spans of text. We introduce a transformer-based architecture that drives a rule synthesis system that leverages a self-supervised approach for pre-training a large-scale language model complemented by an analysis of different loss functions and aggregation mechanisms for variable length sequences of user-annotated spans of text. The results are encouraging and point to different desirable properties, such as speed and quality, depending on the choice of loss and aggregation method.

1 Introduction

Rule-based information extraction is interpretable, maintainable, and highly precise, but typically requires both domain expertise and deep knowledge of an esoteric rule language. The language barrier presents a major impediment to adoption for subject matter experts who are otherwise comfortable with providing examples of their information need in the form of a handful of highlighted spans of text. Synthesizing information extraction rules from such examples has the potential to bridge this divide and empower subject matter experts, but how can we learn to synthesize such programs for any domain? We propose a self-supervised approach for pre-training a large-scale language model for synthesizing information extraction rules using randomly generated rules (programs) paired with matched spans in context (program specifications). The contributions of this work are:

- A transformer-based neural architecture for rule scoring designed to drive a rule synthesis process by jointly encoding candidate rules and a specification that represents the user's intent.
- The introduction of a relevant in domain self-supervised pre-training objective for the rule synthesis problem.
- The exploration of different training scenarios using different loss functions and aggregation methods to score candidate rules in the presence of variable length user specifications.

2 Related Work

Generating computer programs automatically has been a longstanding dream within the field of Artificial Intelligence. The goal of program synthesis is to generate programs from a high-level *specification*[1] (Gulwani et al., 2017). Existing approaches to program synthesis fall into one of two broad categories: search-based methods (e.g., enumerative search, stochastic search (Alur et al., 2013) etc.) and constraint satisfaction (Solar-Lezama, 2009; Torlak and Bodik, 2013). In this work, we focus on search-based program synthesis (Alur et al., 2018), specifically a neural-guided enumerative search.

Neural approaches have come to dominate search-based program synthesis (Balog et al., 2016; Parisotto et al., 2016; Kalyan et al., 2018). Similar to Parisotto et al. (2016), our approach learns a distribution over programs to guide every step of the search. Unlike Parisotto et al. (2016), however, we score only the transitions allowed by the DSL grammar and encode the specification in a manner which reflects that our programs are meant to match natural language.

Rule-based information extraction systems are more interpretable than its statistical counterparts.

[1]A description (visual, example-based, etc.) of what the program should accomplish.

Rule languages with high expressiveness allow to model complex surface and syntactic patterns (Valenzuela-Escárcega et al., 2016). These systems are suitable to create highly specialized domain-specific information extraction tools without the need of large and expensive annotated datasets (Valenzuela-Escárcega et al., 2018). Unfortunately, mastering rule-based systems often implies a steep learning curve and a significant time investment by domain experts.

In this work we focus on the intersection of both pattern-based and neural-based techniques by training a statistical model to *synthesize* rule patterns by exposing it to user-provided examples.

3 Odinson Information Extraction System

In this work, we are interested in synthesizing information extraction rules expressed in a domain specific language (DSL) first described in Valenzuela-Escárcega et al. (2016). This language supports extraction rules based on token constraints (e.g., parts-of-speech and lemmas) as well as syntactic patterns. However, only surface rules are targeted in this work, leaving support for syntactic rules for future work.

We apply the rules expressed in the DSL using the Odinson information extraction framework (Valenzuela-Escárcega et al., 2020), which supports the efficient application of the extraction rules over a large corpus through the use of a custom Lucene[2] index. Like Lucene, Odinson can store an index on-disk or in-memory, and we take advantage of both indexes types during this work. An on-disk index is used to store a large corpus used during the data generation process described in Section 5, and an in-memory index is created on-the-fly during the enumerative search to store the sentences that form part of the user *specification*, indicated in Algorithm 1 as the $make\_index()$ function.

4 Enumerative Search

The enumerative search procedure, outlined in Algorithm 1, takes as input a *user specification*, (referred to as specification for brevity) comprised of a collection of sentences and a set of spans representing the desired extractions; a *rule scorer* component which drives the behavior of the search; and a *performance threshold* that functions as a stopping criteria for the search.

The specification encodes the user's intent and is the main source of signal at the time of scoring any given partial rule encountered during the search.

Rules are composed of syntactic elements defined by the DSL. One noteworthy element is the *placeholder*, represented by the symbol $\square$, which is introduced to represent an underspecified portion of the rule that must be expanded by following the grammar of the DSL. A rule is considered *valid* (grammatical) if it has no placeholders, otherwise it is a *partial* rule containing one or more placeholder. Partial rules are subject to further expansions of the form allowed by the DSL grammar.

To conduct an enumerative search, we create an in-memory index containing the phrases included in the rule's specification and initialize a priority queue with a partial rule consisting of a single placeholder (i.e., the root of the search tree). At each iteration, we retrieve the top-ranked partial rule in the priority queue and check its validity. If it is a valid rule, we query the index to verify if the rule matches the spans highlighted in the specification, and compute a score to measure the rule's performance (e.g., an F1 score over the spans matched in the specification). If the score is above some threshold[3], the candidate rule is returned. If the rule still contains placeholders, we expand the left-most placeholder according to our DSL grammar and use the rule scorer to produce a score for each of the expanded rules with respect to the user-provided specification. Each expanded rule is placed into the priority queue according to its score. The process repeats until a complete rule is found that meets or exceeds the specified performance threshold.

Our enumerative search process implements the branch-and-bound algorithm (Land and Doig, 1960) using a priority queue. An ideal scorer would guide the expansions directly to a correct rule approximating a depth-first search, but the priority queue allows the search to *backtrack* to the most promising candidate if necessary.

It is worth noting that the rule scorer component of the algorithm can be any function that takes as input a partial rule and the specification, and returns a score denoting the rule's priority for further expansion. In the remaining sections of this work, we describe neural architectures for training rule

[2]`https://lucene.apache.org/`

[3]A system hyperparameter.

scoring models (§7.1) with supervised training data (§5).

Algorithm 1 Enumerative search.

```
Require: $spec$                         ▷ user specification
Require: $scorer$                       ▷ partial rule scorer
Require: $threshold$                    ▷ score threshold
  $(sentences, gold\_spans) \leftarrow spec$
  $index \leftarrow make\_index(sentences)$
  $queue \leftarrow priority\_queue()$
  $push(queue, \square, \infty)$
  while $queue \neq \emptyset$ do
      $candidate \leftarrow pop(queue)$
      if $is\_valid(candidate)$ then
          $results \leftarrow query(index, candidate)$
          $score \leftarrow eval(results, gold\_spans)$
          if $score \geq threshold$ then
              return $candidate$
          end if
      else
          $children \leftarrow expand(candidate)$
          for all $child \in children$ do
              $score \leftarrow scorer(child, spec)$
              $push(queue, child, score)$
          end for
      end if
  end while
```

5 Data Generation

To learn to synthesize information extraction rules in a supervised fashion, we first need $(spec, rule)$ pairs. Conceptually, a $spec$ (i.e., matches in context) is easily obtained by querying an index, as long as the $rule$ is available. Our data generation pipeline can be broken down into 4 steps, as highlighted in Algorithm 2.

Algorithm 2 The algorithm to generate $(spec, rule)$ pairs without any supervision

```
1: Generate a random rule $r$
2: Query the index using the rule $r$
3: Select a spec $s$ out of all the query results
4: Return $(s, r)$
```

In order to generate a rule (step 1 in Algorithm 2), we need a sentence and a span of interest.[4]

[4]In practice, this would be provided by a user (e.g., a domain expert). To generate data for our pre-training task in this work, we simply select a random sentence together with a random span within that sentence.

Then, we randomly manipulate constraints to alter the rule's complexity. After each subsequent change in the candidate rule, we query a large index to avoid generating rules without any matches. Once we have a final form of a rule, we use it to query the index and collect the matches (spans in context) to use as the rule's specification (*spec*). We describe our rule generation process in Algorithm 3.

Algorithm 3 The algorithm to generate a random rule

```
Require: Sentence $sent$ together with $span$, a
    span inside $sent$
  $rule \leftarrow$ surface_constraints$(sent, span)$
  $rule \leftarrow$ random_constraints$(rule)$
  $rule \leftarrow$ random_surface$(rule)$
```

The heart of our rule generation algorithm lies inside 3 functions: `surface_constraints`, `random_constraints`, and `random_surface`. Concretely, `surface_constraints` uses the underlying *tag, lemma,* or *word* constraints to generate an initial rule. Then, `random_constraints` adds additional token-level constraints, such as `and`, `or`, and `not`. Lastly, `random_surface` adds surface level constraints, such as wildcards and `or` constraints. For example, consider `The quick brown fox jumps over the lazy dog` together with the desired span *quick brown fox*. The `surface_constraints` will generate an initial rule, such as `[lemma=quick] [tag=JJ] [word=fox]`. Then, `random_constraints` will modify this initial rule and add random constraints, changing the rule to something like `[lemma=quick | lemma=fast] [tag=JJ] [word=fox]`. Then, `random_surface` adds surface level constraints. For example, the rule can now become `([lemma=quick | lemma=fast])? ([tag=JJ] | [word=lazy]) ([word=fox] | [word=cat])`. Note that both `random_constraints` and `random_surface` add a random number of changes to the initial rule.

6 Pre-training

We take inspiration from the Question-Answering community (Rajpurkar et al., 2016, 2018) and train a model to predict the span matched by a given rule in a given sentence. Concretely, we use a dataset generated according to the description in Section 5 and interpret the concatenation of the rule ($rule$) with the sentence ($sent$) as the question. Then, the self-supervised task becomes to predict the start and the end of the span, as in classical span-based question answering.

This pre-training objective has two aims. First, we expose the backbone model (a transformer) to the in-domain vocabulary in which the words and symbols of the DSL are much more frequent than they are in the Wikipedia or Book corpora (Devlin et al., 2019). Our second aim is to expose the model to an unfamiliar task closely related to our ultimate goal: span prediction (i.e., the match for an information extraction rule against some sentence). By learning to predict spans, the model is primed to learn to score candidate rules generated through enumerative search.

7 Rule Scorer Architecture

The heart of the enumerative search process lies at the *rule scoring function*. The rule scoring function assigns a priority value to each of the rule's expansions encountered throughout an enumerative search. Since priority scores control the exploration behavior during the search, it is critical to have an optimized scoring model that can discriminate between promising or futile rule expansions.

We opt to follow a data-driven approach to train a rule-scoring artificial neural network. Figure 1 depicts the architecture of scoring network. The network takes as input a rule: either partially or totally expanded, and the specification; as output, it produces a score used to prioritize a candidate rule in the search process.

We used the rule-specs described in §5 to build a dataset of transitions for rule scoring. We split the rules into training, development and testing with 1.5 million, 6,000 and 500 items, respectively. For the training and development subsets, we used the Odinson's language grammar to enumerate all the transitions necessary to derive each of the rules, and collected the transitions along the path from the root placeholder to the ground truth rule that matches the specification. To obtain negative examples, at any given step we collected the syntactically valid expansions that *don't lead* to the ground truth rule. Figure 2 shows an example of the expansion steps to generate a simple rule composes of two token constrains in DSL. Each row shows a transition, considered a positive training example while the rest of the other syntactically valid transitions (not shown in the figure) are used as negative examples.

Following this approach, the dataset contains 9,731,804 and 748,836 transitions in the training and development subsets, respectively. The ratio of positive to negative transitions is 1 to 3.21.

7.1 Encoding the Rule with the Specification

The provided specification encodes the intent of the rule. Without a specification, it is impossible to know what a rule is expected to match. In other words, the specification sets the context of a rule, and without it, there is no signal to guide the search process beyond the syntactic properties of the DSL's grammar and the statistics of the training dataset.

The specification consists of a variable length list of sentences with an annotated span that the target rule is expected to match. We propose a two-step method to encode a rule with its specification into a combined representation inspired in the investigation of different aggregation methods for multiple input sequences proposed by (Noriega-Atala et al., 2022). This rule-spec embedding is then used to compute the priority score.

In the first step, a partially expanded rule will be paired with each phrase in the spec. The pair is encoded by prepending the rule to the phrase, separated by the `[SEP]` token. Special markup tokens are inserted at the boundaries of the match. This annotated input sequence is then passed to a BERT-based encoder to generate a contextualized representation of the rule with one of its individual spec phrases. Figure 3 shows an example

In the second step, we aim to reduce all the contextualized representations of the partial rule to represent the totality of the user's intent into a fixed-size embedding. We achieve this by collecting the `[CLS]` tokens of the contextualized sequences from the previous step and aggregate them together using one of the following methods: *average*, *max-pooling over time* or an *attention mechanism* (Yang et al., 2016), whose attention matrix and single query vector are tunable parameters optimized during training.

8 Experiments

We tested the rule scoring architecture on several rule synthesis processes. For each model, we control the loss function (§8.1) used for training and

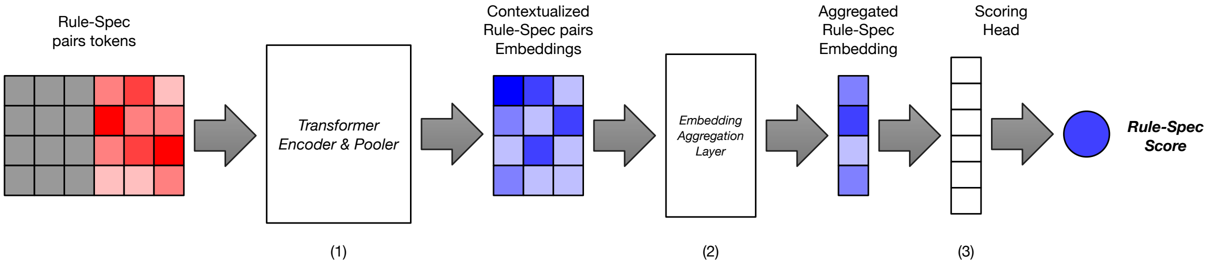
Rule-Spec pairs tokens
Transformer Encoder & Pooler
(1)
Contextualized Rule-Spec pairs Embeddings
Embedding Aggregation Layer
(2)
Aggregated Rule-Spec Embedding
Scoring Head
(3)
Rule-Spec Score

Figure 1: Rule scoring architecture. Step 1 takes as input the pair-wise concatenation of a) the partial rule and b) the annotated phrases in the provided specification. The input is fed through a foundation transformer model which outputs the `[CLS]` embedding for each rule-spec pair. Step 2 aggregates the matrix of `[CLS]` embeddings, either through average pooling or an attention layer, and outputs a fixed-size rule-spec vector. Step 3 linearly maps the rule-spec vector into a real-valued scalar score. The output score will be used as the priority value during the enumerative search process.

$\square \rightarrow \square\square$
$\square\square \longrightarrow [\square]\ \square$
$[\square]\ \square \longrightarrow [\text{tag} = \square]\ \square$
$[\text{tag} = \square]\ \square \longrightarrow [\text{tag} = \textit{JJ}]\ \square$
$[\text{tag} = \textit{JJ}]\ \square \longrightarrow [\text{tag} = \textit{JJ}]\ [\square]$
$[\text{tag} = \textit{JJ}]\ [\square] \longrightarrow [\text{tag} = \textit{JJ}]\ [\text{word} = \square]$
$[\text{tag} = \textit{JJ}]\ [\text{word} = \square] \rightarrow [\text{tag} = \textit{JJ}]\ [\text{word} = \textit{rule}]$

Figure 2: Expansion transitions to generate a rule. The transitions shown here are used as positive examples in the training set for the rule scoring model.

the method for aggregating the specification (§7.1). All models are fine-tuned on top of a BERT checkpoint pre-trained for rule span prediction (§6).

Each trained model was applied on a held-out test set of 500 rule synthesis problems with their respective specification. Each synthesis process was carried out by an enumerative search with a limit of 500 steps.

8.1 Loss Functions for Rule Scoring

One crucial property required to carry out enumerative search efficiently is the rule scoring function. The function must give a high score to rules generated from transitions leading towards a rule that matches the specification and vice-versa.

We explore two loss functions designed to reward accurate transitions and penalize transitions that don't lead towards the ground-truth rule.

8.2 MSE Loss

We use the mean squared error loss function (Equation 1) in which an expansion is scored as $i_i = 1$ when it the expansion is the result of a transition towards the ground-truth rule, and $l_i = 0$ otherwise. This loss configuration does not take into account any information about the location of the expansion in the AST of the rule.

$$\ell(\mathbf{l}, \mathbf{s}) = \frac{1}{n}\sum_{i=1}^{n}(l_i - s_i)^2 \tag{1}$$

8.3 Margin loss

Additionally, we use a margin loss function to train a scoring function to rank the scores of a rule with respect to the score of its parent. The element-wise loss function from Equation 2 takes as as input arguments two scores: A partial rule's score (s_c) and its parent's rule score (s_p). Each score is generated by a forward pass through the model. Individual losses within a batch are averaged to generate the batch's loss. Optimizing this loss will tune the model such that the difference between the pair of scores is at least as large as the margin hyper-parameter m. This loss function is designed to give a higher score to a rule or partial rule than its parent if it was a transition leading towards the ground-truth and vice-versa. Ideally, this property should prioritize partial rules that are closer to matching the specification.

$$\ell(s_p, s_c, m) = \begin{cases} \max(0, m - s_c + s_p) \\ \qquad \text{if } s_c \text{ is correct} \\ \max(0, m - s_p + s_c) \\ \qquad \text{if } s_c \text{ is incorrect} \end{cases} \tag{2}$$

9 Results

Table 1 shows the number of problems for which each rule scoring model matched the user specification. Cases in which all the spans in the specification are matched exactly are considered *exact matches* and those where a span is missing or an incorrect span is matched are considered *partial*

`[CLS]` [tag = *JJR*] [lemma = *natural*] [□|□|□] □ `[SEP]` The **<MATCH>** more natural , or background **</MATCH>** , sound you can tape , the better . `[SEP]`

`[CLS]` [tag = *JJR*] [lemma = *natural*] [□|□|□] □ `[SEP]` so there are **<MATCH>** less natural , wild **</MATCH>**, hatchery fish that make it to the ocean `[SEP]`

`[CLS]` [tag = *JJR*] [lemma = *natural*] [□|□|□] □ `[SEP]` The further back one traces the race , the fewer are concerned in the government ; the fewer are so concerned , the **<MATCH>** more natural , **</MATCH>** because the easiest , is the system of effecting changes - aye , improvements - by " despatching " the government . `[SEP]`

Figure 3: Partial rule and its target specification encoded as input to a rule scoring network. The partial rule is paired with every phrase in the specification to attend to every match with a transformer encoder.The boundaries of the specification's spans are delimited with specially designated markup tokens in the architecture's tokenizer model. For each input sequence, the rule scoring network will pool the `[CLS]` output embedding as the contextualized representation of the partial rule with respect to its corresponding matching span. The matrix of `[CLS]` embeddings will reduced to fixed size rule-spec encoding with an aggregation layer as described in §7.1. The contents of this figure represent the left-most input block in figure 1

Loss Function	***Spec Aggregation***	*Exact Matches*	*Partial Matches*	*Any Matches*
Margin	Attention	68(14%)	263(53%)	331(67%)
Margin	Average	57(11%)	248(50%)	181(61%)
MSE	Attention	**113**(**23**%)	358(72%)	**471**(**95**%)
MSE	Average	84(17%)	**383**(**77**%)	838(94%)
Margin	No Spec	0	28(6%)	28(6%)

Table 1: Number of matches in the specifications of the testing set. Exact matches are cases in which all the spans in the specification are matched exactly. Partial matches are cases where a) a span is missing or b) an incorrect span is matched. Total matches are the sum of both.

matches. The right-most column counts the number of matches, irrespective of the type.

The architectures that aggregate the rule-span pair encodings using an attention mechanism are better at matching the specification in terms of exact and partial matches. Specifically, the model trained using the MSE loss function has the highest exact match rate and combined match rate. For any given problem, every span in the specification represents constraints for rule synthesis. These constraints interact together to encode some intent. We hypothesise that the attention mechanism helps capture the signal in those implicit constraint interactions better than averaging the rule-span pair encodings, which effectively weights the contribution of every encoding to the rule's score equally.

To highlight the crucial role of the specification, we trained a model that generates scores by only encoding partial rules, ignoring the specification during the enumerative search algorithm. This model is represented by the bottom row of the table. Its unsurprisingly poor performance illustrate how without the specification (which represents the intent), there is simply not enough signal to successfully synthesize a rule.

In addition to the number of matches of each model, we are also interested in the quality of the matches. Tables 2 and 3 show the macro and micro average performances of the rule scoring models, respectively. To compute precision, recall and F1 scores, a) matches to specification spans, b) matches to spans not in the specification, and c) unmatched spans in the specification are considered a) true positives, b) false positives, and c) and false negatives, respectively. The models that encode the rule and specification with attention outperform those that average. The attention mechanism also improves recall. These results are consonant with our hypothesis about the utility of the attention mechanism to model the interaction of the elements in the specification. In addition, models trained with the margin loss function are faster at synthesizing rules (i.e., they require fewer steps to generate a complete rule). This observation reflects the design, as the margin loss prioritizes partial rules close to complete expansion, resembling more a depth-first search approach, whereas training with MSE just estimates how "right" or

Loss Function	*Spec Aggregation*	*Precision*	*Recall*	*F1*	*Steps*
Margin	Attention	0.55 ± 0.02	0.34 ± 0.02	0.36 ± 0.02	$\mathbf{13.08 \pm 0.37}$
Margin	Average	0.49 ± 0.02	0.32 ± 0.02	0.33 ± 0.02	13.43 ± 0.46
MSE	Attention	$\mathbf{0.82 \pm 0.01}$	$\mathbf{0.50 \pm 0.02}$	$\mathbf{0.56 \pm 0.02}$	74.15 ± 3.65
MSE	Average	0.73 ± 0.02	0.45 ± 0.02	0.48 ± 0.01	55.55 ± 3.73
Margin	No Spec	0.01 ± 0.00	0.02 ± 0.01	0.01 ± 0.00	4.0 ± 0.0

Table 2: Macro-average performance of different rule scoring models on the testing set. Results are computed by evaluating the rules generated using enumerative search on their corresponding specification in the testing set (see §7). The testing dataset is bootstrapped re-sampled 10,000 times to calculate standard deviations of each metric. To compute precision, recall and F1 scores, a) matches to specification spans, b) matches to spans not in the specification, and c) unmatched spans in the specification are considered a) true positives, b) false positives, and c) false negatives, respectively. The steps column reports the average number of steps to successfully find a rule.

Loss Function	*Spec Aggregation*	*Precision*	*Recall*	*F1*
Margin	Attention	0.55 ± 0.03	$\mathbf{0.41 \pm 0.03}$	$\mathbf{0.47 \pm 0.02}$
Margin	Average	0.53 ± 0.03	$\mathbf{0.41 \pm 0.03}$	0.46 ± 0.03
MSE	Attention	$\mathbf{0.69 \pm 0.04}$	0.30 ± 0.02	0.42 ± 0.03
MSE	Average	0.32 ± 0.02	0.27 ± 0.02	0.30 ± 0.01
Margin	No Spec	0.13 ± 0.03	0.16 ± 0.04	0.14 ± 0.03

Table 3: Micro-average performance of different rule scoring models on the testing set. Results are computed by evaluating the rules generated using enumerative search on their corresponding specification in the testing set (see §7) and the matches to calculate micro average performance metrics. The testing dataset is bootstrapped re-sampled 10,000 times to calculate standard deviations of each metric. Metrics are compared similarly. The definitions of precision, recall and F1 are the same of table 2

"wrong" is the rule, and has no implicit consideration about how far or close a partial rule is to completing expansion.

We looked at the rules generated using the MSE+Attention rule-scoring model. Out of all the rules generated, approximately 2.5% perfectly match the gold rule. Nevertheless, approximately 20% of the generated rules obtain a perfect F1 score (i.e. 1.0). This is an example of *program aliasing*, where a different program produces the same result. In our case specifically, a different rule matches the same specification. We can observe this phenomenon in the first two rows of table 4. Nonetheless, there is still ample room for improvement, as we can see how sometimes the synthesizer generates a rule that misses most of the specification.

10 Future Work

The results presented in this work are promising and open the door for further research in several directions that will help to better understand the properties of the rule scoring architectures. In its current state, our approach has at least avenues to future work.

Extrinsic Analysis We evaluated the methods on a testing dataset. This evaluation is useful to compare the relative performance of the different architectures analyzed in this work. It is necessary to implement named captures to be able perform an *extrinsic* evaluation, similar to the evaluation protocol proposed by (Vacareanu et al., 2022) on an external dataset to validate the utility of our method for the information extraction task.

Finding equivalent rules Our training procedure prunes branches that will not lead to the target rule, but this may inadvertently prune paths leading to equivalent rules that match the same specification. The large search space makes it impractical to enumerate all equivalent rules for each specification. Reinforcement learning is a viable alternative to training a rule scoring model. Multiple rules that

Target Rule	*Synthesized Rule*	*F1*
`[tag=NN] [lemma=in] [raw=many] [word=of] [tag=PRP$]`	`[tag=NN] [lemma=in] [lemma=many] [lemma=of] [tag=PRP$]`	1.0
`[raw=always] [lemma=make] [tag=NNS] [word=that]`	`[lemma=alway] [tag=VBP] [lemma=decision] [lemma=that]`	1.0
`[lemma=describe] [raw=how] [word=the]`	`[tag=VBZ] [lemma=how] [tag=DT]`	0.59
`[lemma=mikhail | tag=NN]`	`[word=The | lemma=range]`	0.08

Table 4: Examples of rules generated with an enumerative search. Target Rule is the ground truth rule which matches the user specification. Synthesized Rule is the rule generated by the enumerative search process; F1 is the synthesized rule's score over the user specification for the corresponding target rule.

match the specification can be found by exploring the space of syntactically correct expansions. This could improve the expressiveness of a rule synthesis system and increase data efficiency without incorrectly penalizing valid rules that are not part of the training dataset in supervised learning.

Grammar synthesis The current approach assumes a single rule must match the entire specification. For diverse specifications (i.e., spans with highly varied contexts), a single rule may end up containing many disjunctive clauses. Long, complex rules may adversely affect interpretability and maintainability. Rather than generating a single rule for a specification, it may be advantageous in some cases to learn to generate a set of two or more complementary rules with low complexity and better generalization.

Interactive use The model trained using margin loss and an attention mechanism for encoding is able to find a rule much faster (on average) than the rest of the models without suffering a steep loss in performance. Further investigation is needed to understand not only what changes might increase the quality of the matches in rule synthesis but also what changes will make they system fast enough for interactive use.

References

Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design*, pages 1–8.

Rajeev Alur, Rishabh Singh, Dana Fisman, and Armando Solar-Lezama. 2018. Search-based program synthesis. *Communications of the ACM*, 61(12):84–93.

Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2016. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989*.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota. Association for Computational Linguistics.

Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. 2017. Program synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119.

Ashwin Kalyan, Abhishek Mohta, Oleksandr Polozov, Dhruv Batra, Prateek Jain, and Sumit Gulwani. 2018. Neural-guided deductive search for real-time program synthesis from examples. *arXiv preprint arXiv:1804.01186*.

AH Land and AG Doig. 1960. An automatic method of solving discrete programming problems. *Econometrica: Journal of the Econometric Society*, pages 497–520.

Enrique Noriega-Atala, Peter M. Lovett, Clayton Morrison, and Mihai Surdeanu. 2022. Neural architectures for biological inter-sentence relation extraction. In *Scientific Document Understanding 2022*, number 3164 in CEUR Workshop Proceedings.

Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. 2016. Neuro-symbolic program synthesis. *arXiv preprint arXiv:1611.01855*.

Pranav Rajpurkar, Robin Jia, and Percy Liang. 2018. Know what you don't know: Unanswerable questions for squad. In *ACL*.

Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. 2016. Squad: 100,000+ questions for machine comprehension of text. In *EMNLP*.

Armando Solar-Lezama. 2009. The sketching approach to program synthesis. In *Asian Symposium on Programming Languages and Systems*, pages 4–13. Springer.

Emina Torlak and Rastislav Bodik. 2013. Growing solver-aided languages with rosette. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, pages 135–152.

Robert Vacareanu, Marco A. Valenzuela-Escárcega, George Barbosa, Rebecca Sharp, and Mihai Surdeanu. 2022. From examples to rules: Neural guided rule synthesis for information extraction. In *Proceedings of the 13th Language Resources and Evaluation Conference (LREC)*.

Marco A. Valenzuela-Escárcega, Gus Hahn-Powell, and Dane Bell. 2020. Odinson: A fast rule-based information extraction framework. In *Proceedings of the 12th Language Resources and Evaluation Conference*, pages 2183–2191, Marseille, France. European Language Resources Association.

Marco A. Valenzuela-Escárcega, Gus Hahn-Powell, and Mihai Surdeanu. 2016. Odin's runes: A rule language for information extraction. In *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC'16)*, pages 322–329, Portorož, Slovenia. European Language Resources Association (ELRA).

Marco A Valenzuela-Escárcega, Özgün Babur, Gus Hahn-Powell, Dane Bell, Thomas Hicks, Enrique Noriega-Atala, Xia Wang, Mihai Surdeanu, Emek Demir, and Clayton T Morrison. 2018. Large-scale automated machine reading discovers new cancer-driving mechanisms. *Database*, 2018. Bay098.

Zichao Yang, Diyi Yang, Chris Dyer, Xiaodong He, Alex Smola, and Eduard Hovy. 2016. Hierarchical attention networks for document classification. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 1480–1489, San Diego, California. Association for Computational Linguistics.