EBREG-RL: EXAMPLE-BASED REGULAR EXPRESSION GENERATOR VIA REINFORCEMENT LEARNING

A PREPRINT

Dmitry Beresnev
AIDS-MS1, Innopolis University
d.beresnev@innopolis.university

Vsevolod Klyushev
AIDS-MS1, Innopolis University
v.klyushev@innopolis.university

Nikita Yaneev
AIDS-MS1, Innopolis University
n.yaneev@innopolis.university

1 Introduction

Nowadays big-tech companies increasingly prefer data-driven development, which requires careful extraction of information, including from unstructured or almost unstructured text. This task is considered as extremely tough, but in many real-world scenarios even unstructured text information have underlying syntactic pattern that can be characterized by a regular expressions, due to their adaptability and expressiveness. However, even for experienced programmers, writing RegEx by hand is a boring, time-consuming, and error-prone process. Moreover, despite widespread use of regular expression, there are not so many researches and literature on the automatic RegEx generation. That is why we chose an example-based RegEx generation as a topic of the project of this course.

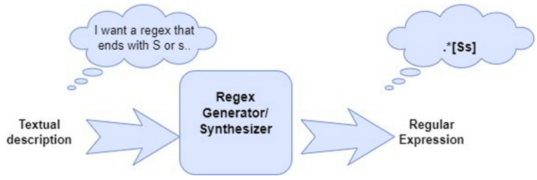

Figure 1: NL2RE pipeline

One approach [**?** **?**] uses LLMs to convert natural language prompts into regular expressions (NL2RE). This method offers flexibility, allowing the model to generate a wide variety of RegExes. However, it requires significant computational resources and careful architecture design. This method is not ideal for generating RegExes that are specifically tailored to a particular domain or task. It is also challenging to ensure that the generated RegExes are accurate and efficient.

Another approach [**?** **?**] uses genetic programming (GP) to generate regular expression for specific tasks based on labeled data. The GP algorithm searches for the best-performing regular expression by iteratively evolving a population of candidate solutions. This method requires careful design of hyperparameters, such as the size of the population and the mutation rate, as well as a well-defined fitness function to evaluate the performance of each regular expression.

2 Problem formulation

We propose a new approach that leverages reinforcement learning (RL) to generate regular expression. This method formulates the regular expression generation process as a Markov decision process (MDP) with fully-observable deterministic episodic environment.

Each action corresponds to a regular expression syntax unit, state space would be a set of all possible sequences of syntax units of fixed length. For reward function various information retrieval metrics can be used as indicators of how good the generated regular expression is.

2.1 Reverse Poland Notation

Due to the fact that regular expressions have operations with different precedence and number of arguments, we decided to use the Reverse Polish Notation (RPN) for the construction of regular expressions.

In our RPN implementation we have 101 different actions:

- 93 tokens (letters, difits, quantifiers, finish token, etc.),
- 2 binary operations ("concat" and "|"),
- 3 unary operations ("*", "+", "?"),
- 3 many to one operations ("[]", "[̂]", concat_all").

2.2 Action Space

Action space for our RL agent consists of all possible actions for RPN.

2.3 State Space

The state space consists of n placeholders, where each placeholder can either contain an action from the action space or remain empty (denoted by a special empty token). To derive the current regular expression (state), we iterate over the non-empty tokens and apply the corresponding actions using Reverse Polish Notation (RPN).

2.4 Dataset

...

2.5 Reward

Reward function is a major part of RL pipeline. We come up with 2 approaches for reward calculation. Note, that we will calculate such rewards only by the end of the episode. Before that we will give 0 as reward. Let's start with common parts.

First, we start with a text from which we want to extract specific information. To do this, we need to: Identify the indices of the characters we want to retrieve. Construct a "target bit mask" of length n (where n is the text length), where:

- 1 marks positions to be included.
- 0 marks positions to be excluded.

Then, we get the current regular expression from RPN and apply it to the text, retriving indices of founnd symbold. After that, we construct the "current bit mask" in the same way, as "target bit mask"

2.5.1 Simple approach

First idea of the reward is the following:

$$\alpha \cdot \sum (A \oplus B) + \beta \cdot |C - D| + \gamma \cdot E$$

where:

- A - "target bitmask"
- B - "current bitmask"
- C - number of target words
- D - number of words, retrieved with regexp
- E - token length of regexp

2.5.2 Complex approach

Second idea of the reward is based on the fact, that we can calculate accuracy, precision, recall and F1 on bitmasks. Thus reward function would be the following:

$$\alpha \cdot F1 + \beta \cdot P + \gamma \cdot R + \delta \cdot |C - D| + \omega \cdot E$$

where:

- F1 - F1 score on bit masks
- P - Precision on bit masks
- R - Recall on bit masks
- C - number of target words
- D - number of words, retrieved with regexp
- E - token length of regexp

3 RL approaches

We decided to check 2 RL approaches - Actor-Critic and Reinforce.

3.1 Actor-Critic

For Actor-Critic approach we define Neural Network (NN) with two heads: one for policy and one for state value. The training procedure is the following:

Algorithm 1 Actor-Critic

Input: empty state of size n, environment env
$done \leftarrow False$
$actions \leftarrow []$
$states \leftarrow []$
$rewards \leftarrow []$
$state \leftarrow env.reset()$
while $!done$ **do**
$logits \leftarrow net(state)[0]$
$action \leftarrow choose\_action(logits)$
$new\_state, reward, done \leftarrow env.step(action)$
$actions, states, rewards \leftarrow action, state, reward$
end while
$cumulative\_rewards \leftarrow accomulate(rewards)$
$logits, values \leftarrow net(states)$
$loss\_val \leftarrow MSE(values, cumulative\_rewards)$
$log\_probs \leftarrow LogSoftmax(logits)$
$adv \leftarrow cumulative\_rewards - values$
$log\_probs\_actions \leftarrow adv \cdot log\_probs$
$loss\_policy \leftarrow -mean(log\_probs\_actions)$
$probs \leftarrow Softmax(logits)$
$entropy \leftarrow mean(sum((probs \cdot log\_probs), dim = 1))$
$entropy_loss \leftarrow \beta \cdot entropy$
$loss\_policy.backward()$
$loss\_val \leftarrow entropy_loss + loss\_val$
$loss\_val.backward()$
return net

3.2 Reinforce

...

4 Experiments

We decided to check our models on 3 different experiments:

- Digits retrieval
- Retrieval of words, that consists of certain characters
- Email addressed retrieval

5 Results and Discussion

...

6 Conclusion

...

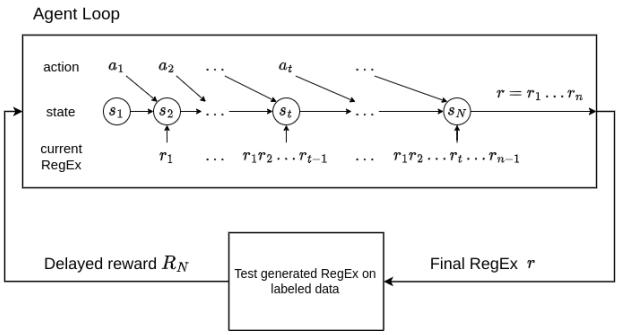

Figure 2: our pipeline

Figure 3: zhopa

7 Citation examples

[?] [?] [?] [?]