Regex-RNN: Generating Regular Expressions from Natural Language with Recurrent Neural Nets

Nicholas Locascio
MIT
njl@mit.edu

Eduardo De León
MIT
edeleon4@mit.edu

Abstract

This work presents a neural architecture for learning how to translate natural language into regular expressions. We use a dataset of 824 natural language and regular expression pairs [1] and train a deep recurrent neural net to generate regular expressions from natural language prompts. This model is trained end-to-end with little to no feature engineering. We evaluate our model by comparing our generated regular expressions to our answers using DFA equivalence [2] and achieve 56.6% accuracy on our dataset. Code hosted here: https://github.mit.edu/njl/regex-rnn

1 Introduction

Regular Expressions are an incredibly powerful tool with a variety of useful applications. Unfortunately, they are often brittle, can be hard to learn, and are most certainly hard to write. It would be beneficial then, if humans were able to specify their needs in natural language and an automated system could generate the corresponding regular expression. This project aims to learn a system capable of such a task.

1.1 Motivation

Regular Expressions have many applications in searching, parsing, and matching text. However, the syntactical representation of Regular Expressions is rather obtuse, being neither particularly readable nor robust. Most Regular Expressions, in software engineering practice, have to be thoroughly commented to be properly read and understood. These natural language descriptions are much easier for humans to read, write, and understand. Our works's goal is to accurately generate a Regular Expression from the descriptive comment that would normally be paired with the Regular Expression.

1.2 Related Work

This work is inspired by the paper Using Semantic Unification to Generate Regular Expressions from Natural Language by Kushman et al [1]. Kushman gathered a dataset of 824 natural language and corresponding regex pairs, and trained a log-linear language model to generate regular expressions from natural language.

Kushman's work focused a lot on techniques for lexical splitting, semantic unification, alignment, and a host of other regex-specific feature engineering. This yielded state-of-the-art results, but we want to achieve comparable results without all of the problem-specific engineering that went into performing semantic unification on regexes. Our system aims to be both general and performant and for this reason, we chose to use deep neural nets.

2 Methods

2.1 Approach

We modeled the problem of predicting regular expressions as a character generation problem. Our model is concerned with predicting the next character in the regex sequence. This means that our model predicts one character at a time, conditioned on the natural language prompt and the preceding regex fragment. To this end, our model must predict the following probability distribution:

$$P(char_i|prompt, current), for\ char_i \in C \tag{1}$$

where

- $char_i$ = regex character to consider generating.
- $prompt$ = natural language prompt.
- $current$ = regex characters generated so far.
- $C =$ the set of potential regex characters to predict.

An example of this would be $P(char_i|$'lines containing dog', '.*d'$), for\ char_i \in C$ = {"o":-1.5, "Z":-1.9, ...}.

2.2 Data Preparation

We split our 824 regex and prompt pairs into 75% train, 25% test sets. We add a special START and END word to every sentence, and add a special START and END char for every regex string. We then represent regular expression characters and natural language words as one-hot vectors.

Since we are modeling the probability of predicting the next character given a natural language prompt and a regular expression fragment, we must construct our training examples in such a way.

We take each regex example and bootstrap it into many examples by making a training example from each character. So an example consists of an input of the natural language prompt and a regex fragment, with an output of the next regex character that succeeds the fragment. This bootstrapping process is shown in Figure 4.

After this bootstrapping process we had 11,355 training examples and 4,004 test examples. By first splitting the regex/prompt pairs, we ensure that our natural language prompts in our test set are never seen before in the training set. This would not be true if we split our dataset randomly after our bootstrapping process.

2.3 Model Architecture

Our Model architecture consists of two parallel RNN pipelines feeding into a ANN pipeline. The first RNN pipeline processes regular expressions. The first layer is an embedding layer that maps each of the one-hot encoding of the natural language words input into a list of 50-dimensional word-vectors. These word-vectors get processed by a 2-layer deep LSTM to output at 256 dimensional natural language sentence thought vector. The other RNN pipeline, similarly, uses the same architecture of an embedding layer and 2 stacked LSTM's. The weights of these pipelines are not shared and learned independently.

Each of these RNN pipelines produce sentence and regex thought vectors, respectively. We then concatenate these thought vectors and feed them into 2 stacked fully connected layers. We then feed our final fully connected layer into a softmax layer which computes the log-likelihood probabilities across all possible generated characters.

2.4 Learned Embeddings for Regex Chars

We ran word2vec on our regular expressions dataset to learn embeddings for our characters. Our word vectors are 50 dimensional, and were able to capture some semantics of the characters. For example, the following word-vector analogies were learned: capitalization, number order, number magnitude. We also visualized these word vectors using t-SNE in Figure 2.

Figure 1: Architecture of Regex-RNN

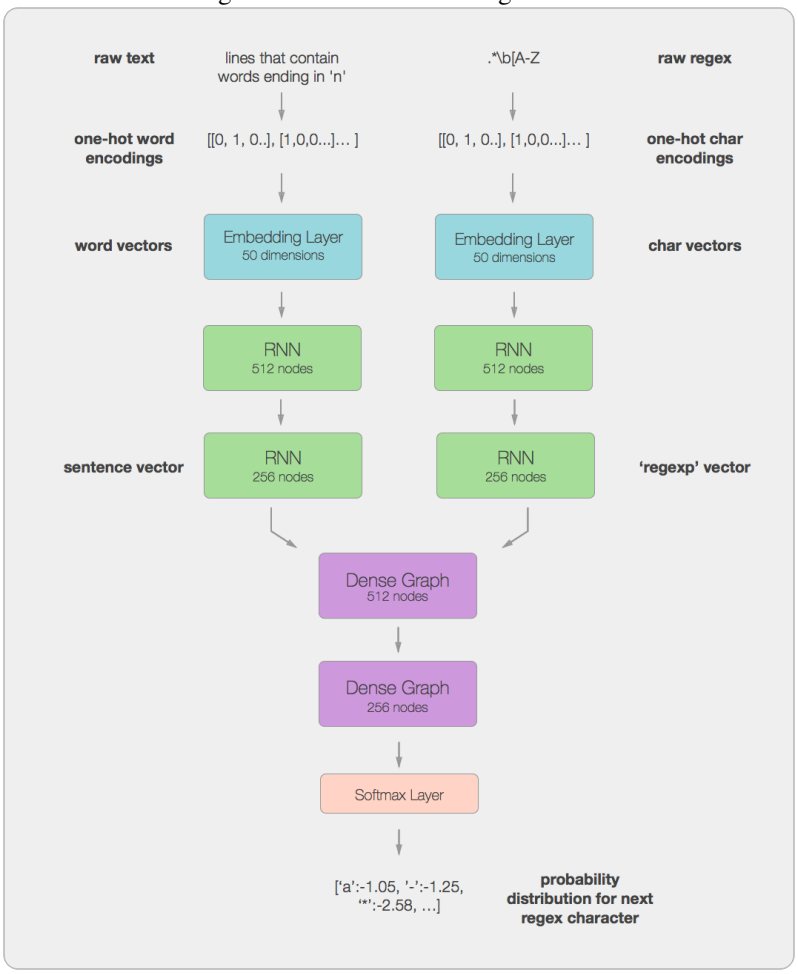
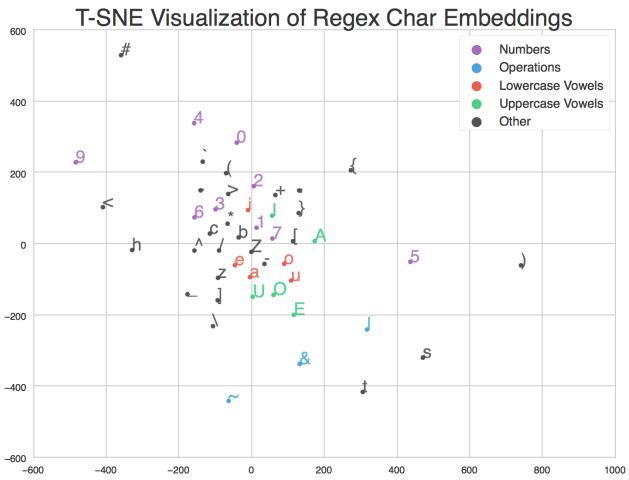


Figure 2: Learned Embeddings for Regex Chars



2.5 Training

We trained our model for 35 epochs using a single AWS GPU. Training took 6 hours, and due to our small dataset, performance converged quickly. In figure 3, we plot the test accuracy and other metrics as a function of epoch number. We were able to achieve 88% accuracy on a per-character level basis.

Figure 3: Training Metrics

3 Beam Search

Our RNN model predicts next character probabilities. However, we'd like to produce full regular expressions in response to the natural language prompt. In order to do this, we must predict the first character of the regex, feed that into our model and then predict the second character, feed those in to the model and so forth until we generate a STOP character. This process creates a path of choices that produces a full regular expression. We compute the probability of this path with:

$$P(path) = \prod_{char \in Path} (P(char|prompt, so\_far)) \tag{2}$$

$$logP(path) = \sum_{char \in Path} log(P(char|prompt, so\_far)) \tag{3}$$

We'd like to consider all paths and choose the one of highest probability, however this graph exploration explodes to be exponential so the problem becomes intractable. On the other hand, we can take the greedy path of highest probability for each step to achieve very fast computation, however, this in practice yields very poor results.

We compromise these two by using a Beam search to explore the space, keeping track of the top 80 path hypothesis per step of the BFS. Each time we consider generating a character at a longer length, we prune our hypotheses down to 80 before continuing the search. This allows us to compute a near-optimal answer in a reasonable amount of time.

3.1 Path Length Normalization

Since we are considering variable-length sequences, we have to normalize our path probability by the path length to get average log probability per character. This normalization essentially computes the average probability of generating each character.

$$logP(path) = \frac{\sum_{char \in path} log(P(char|prompt, so\_far))}{|path|} \tag{4}$$

Without this normalization, our algorithm would invariably prefer shorter strings. Consider the example of "all lines ending in the letter 'f'". The correct regexp for such an example would be '.*f'. When generating this regex, we consider many solutions of varying lengths. If we are comparing between the solution "." vs ".*f". Without length normalization, we incorrectly compute '.' as the answer.

$$logP(.) = -1.50 \tag{5}$$
$$logP(.*f) = (-1.50) + (-0.9) + (-0.6) = -3.0 \tag{6}$$
$$logP(.*f) < logP(.) \tag{7}$$

However, with length normalization, we get the proper prediction of '.*f':

$$logP(.) = -1.50 \tag{8}$$
$$logP(.*f) = \frac{(-1.50) + (-0.9) + (-0.6)}{3.0} = -1.0 \tag{9}$$
$$logP(.*f) > logP(.) \tag{10}$$

We see that with length-normalization, logP(.*f) $>$ logP(.) and we correctly determine that the answer is '.*f'. This normalization allows us to deal with paths of varying lengths, however it is not without its faults which are discussed in 4.2.

4 Discussion

4.1 Evaluation of Results

Our model's accuracy for DFA equivalence is 56.6%. A full detailing of percentages is shown in Figure 4. Our results are 8.9% below the state of the art (Kushman). However, this accuracy is without the crucial pre-processing step that Kushman performs, modifying the training regular expressions such that they factorize in a way that facilitates a direct mapping to the natural language description. Without the pre-processing step, the previous state-of-the art accuracy was at 36.5%. Although lower than the state of the art, our results are promising due to the lack of feature engineering and the surprising success of neural architectures despite a very small data set.

In the Comparison Chart (Figure 4), we compare results for two metrics. DFA equal means a regex was only correct if its FSM was exactly equivalent to the answer regex. Example equal means a regex was only counted correct if it matched all 10 positive and negative examples perfectly.

Figure 4: Comparison of Results

	Semantic Unification (Kushman , 2013)	UBL Model (Kwiatkow ski, 2010)	Regex-RNN Top Result	Regex-RNN Top 5 Results*
DFA Equal	65.5%	36.5%	56.6%	66.5%
Example Equal	x	x	60.6%	70.0%

4.2 Analysis of Successes and Mistakes

Many of our model's successes and mistakes fall into some broad categories. Common mistakes detailed in Figure 5.

Class A mistakes are a result of issues with our Beam-Search path-normalization technique. In the given example, the model generates ' $(.*[A$' perfectly. Consider two possible next predictions, '-', and 'E'. The correct prediction of '$-$' has a higher probability than the prediction of 'E'. However,

Figure 5: Examples of mistakes

Natural Language Prompt	Prediction	Answer	Key Point	
lines that have no letters	~(.*[AEIOUaeiou].*)	~(.*[A-Za-z].*)	Can get lost in thought w/ common sequences	X
lines where there are three characters between instances of "ABC" and "WEX"	.*ABC.*.{2}.*WEX.*|.*WEX.*.{3}.*ABC.*	.*ABC.*.{3}.*WEX.*|.*WEX.*.{3}.*ABC.*	Can mix-up numbers.	X
lines having words with 'ro'.	.*((\b[A-Za-z]+\b)&(.*ro)).*	.*\b[A-Za-z]*ro[A-Za-z]*\b.*	Ambiguous Prompt!	X

once the model predicts AE, it can be pretty certain that it is now is generating vowels, so the probability of generating an I is very high, then a E,O,U,a,e,i,o,u. These multiple high probability predictions drown out the initial low probability prediction of E, so the probability of the incorrect regex is higher than the probability of the correct regex.

Class B mistakes are a result of our embedding space placing numbers char vectors close together. Unfortunately it makes it sometimes hard to distinguish which number is correct.

Class C mistakes are a result of differing interpretations of an ambiguous prompt. Our model interprets the object of the 'with' to be 'lines' whereas the original regex writer interpreted the object of the 'with' to be 'words'. Because of this, we generate a different regex than the golden answer.

4.3 Effect of Training Size on Accuracy

As noted, our model is extremely data-hungry. We ran our model on 75% 25% split and achieved 56.6% accuracy. We also ran our model with the opposite 75%/ 25% test/train split and achieved 11.2%. This stark difference shows that our model improves dramatically with additional data. We will run additional tests with different splits to see the curve, but this initial results suggest more data leads to a much better accuracy.

5 Applications

Currently our model generates regex metacharacters $\backsim$ and & , representing NOT and AND operations, respectively. These operators are not supported in most modern regex representations. Although this limits current usage of our model, a post-processor can be made to translate outputted regular expressions into a compatible representation.

With our model, a system can be made to aid developers in writing regexes. Current practices for debugging a regex involve online editors. Developers give as input a regex and example text and receive as output classifications of the regex on the example text. Our system could instead have the user input a description of the regex along with positive and negative examples. Our model can generate the top 5 results of highest average log likelihood that also properly classify the examples. Although our initial prediction may not be the correct, it can serve as a base for the desired regex. Developers can then modify the regex until they find the correct one. This process can save developers time and improve accuracy when writing regexes.

6 Further Research

1) Use larger dataset. Neural nets are data-hungry models and 824 examples is extremely small.
2) Pre-Process regexes for best DFA alignment.
3) Use Sequence-To-Sequence RNN. We're currently throwing away all representation of why we generated the previous character.

4) Learn shared representations for some words and characters.
5) Use Attention-based NN. Our problem is extremely sensitive to context, so attention-based models would help.
6) Use EM algorithm to compute word-alignments.

Acknowledgments

We would like to thank Regina Barzilay for supporting this work. Further thanks to Nate Kushman for his help and consultation for DFA equivalence techniques. Thanks to Karthik Narasimhan for helpful discussions and suggestions.

References

[1] N. Kushman, R. Barzilay. "Using Semantic Unification to Generate Regular Expressions from Natural Language". North American Chapter of the Association for Computational Linguistics (NAACL) 2013.
[2] Tom Kwiatkowski, Luke Zettlemoyer, Sharon Goldwater, and Mark Steedman. 2010. Inducing probabilistic ccg grammars from logical form with higher-order unification. In Proceedings of EMNLP. [3] J.E. Hopcroft, R. Motwani, and J.D. Ullman. 1979. Introduction to automata theory, languages, and computation, volume 2. Addison-wesley Reading, MA.