SPLITREGEX: FASTER REGEX SYNTHESIS FRAMEWORK VIA NEURAL EXAMPLE SPLITTING

Anonymous authors
Paper under double-blind review

ABSTRACT

Due to the practical importance of regular expressions (regexes, for short), there has been a lot of research to automatically generate regexes from positive and negative string examples. A basic idea of learning a regex is a search-and-repair; search for a correct regex and repair it if incorrect. The problem is known to be PSPACE-complete and the main issue is to obtain a regex quickly within a time limit. While classical regex learning methods do not perform well, recent approaches using deep neural networks show better performance with respect to the accuracy of the resulting regexes. However, all these approaches including SOTA models are often extremely slow because of the slow searching mechanism, and do not produce desired regexes within a give time limit.

We tackle the problem of learning regexes faster from positive and negative strings by relying on a novel approach called 'neural example splitting'. Our approach essentially split up example strings into multiple parts using a neural network trained to group similar substrings from positive strings. This helps to learn a regex faster and, thus, more accurately since we now learn from several short-length strings. We propose an effective regex synthesis framework called 'SplitRegex' that synthesizes subregexes from 'split' positive substrings and produces the final regex by concatenating the synthesized subregexes. For the negative sample, we exploit pre-generated subregexes during the subregex synthesis process and perform the matching against negative strings. Then the final regex becomes consistent with all negative strings. SplitRegex is a divided-and-conquer framework for learning target regexes; split (=divide) positive strings and infer partial regexes for multiple parts, which is much more accurate than the whole string inferring, and concatenate (=conquer) inferred regexes while satisfying negative strings. We empirically demonstrate that the proposed SplitRegex framework substantially improves the previous regex synthesis approaches over four benchmark datasets.

1 INTRODUCTION

Regular expressions (regexes in short) are a powerful tool on processing sequence data in NLP or information retrieval. A regex—a finite sequence—is a formal representation of (infinite) symbol sequences. However, it is not easy to write a good regex for a set of sequences. Especially for those who have little experience on formal expressions, a single mistake in regex composition leads to an invalid regex or, even worse, a regex with different semantics to ones' intention. An automatic regex synthesis tries to moderate this human error.

Examples (Oncina & García, 1992; Bartoli et al., 2014) and natural language descriptions (Locascio et al., 2016; Zhong et al., 2018b) are the two major sources of regex synthesis. Examples are specific in their meaning without any implicit semantics, but are hard to provide the complete user intent described by a regex. On the other hand, natural language descriptions are ambiguous but describes the high-level concept of a regex. As these two different types of information are complementary to each other, there have been many studies (Chen et al., 2020; Li et al., 2021; Zhong et al., 2018a) that suggest to use both modalities as they can use examples to fix the ambiguous expressions generated from natural language descriptions.

The *regex synthesis problem* is to make a proper regex E from both positive and negative sets such that all the positive samples are generated by E and all the negative strings are not generated by E. This problem is also called the regex *inference* or *learning* problem in the literature.

The traditional approaches for the regex synthesis problem are often inapplicable in practical settings. For example, Angluin's (1987) L^* is impractical because it requires the equivalence test between a target regex and a generated regex, which is known to be PSPACE-complete and thus intractable (Stockmeyer & Meyer, 1973). Gold (1978) shows that the state characterization approach is NP-complete and thus intractable. Heuristic approaches based on state-merging DFA induction algorithms such as RPNI (Oncina & García, 1992) and blue-fringe (Lang et al., 1998) are not very promising on synthesizing complex regexes because it is almost impossible to obtain a concise regex even when they are able to produce a concise DFA from examples. As recent approaches, both AlphaRegex (Lee et al., 2016) and RegexGenerator++ (Bartoli et al., 2014; 2016) generate candidate regexes and match against the given examples until the target regex matches all examples successfully. While these approaches generate correct regexes when a time limit is very long, they often fail when the example size is large. In addition, both approaches do not use any characteristics in the given examples, which can be a useful feature to speed up the process. In other words, these methods are impractical for generating regexes within a reasonable time limit.

For practical usage, we notice that the divide-and-conquer approach is promising for synthesizing formal representation. For example, Alur et al. (2017) synthesize a program with hierarchical partition of the given input-output pairs. A decision tree reflects the hierarchical partition forms the control flow of the synthesized program. This method noticeably improves program synthesis time due to the partitioned problem space. Farzan & Nicolet (2021a;b) propose an effective method that converts a program into an equivalent program by splitting a problem into several subproblems. This divide-and-conquer approach is efficient since individual tasks for solving subproblems can run simultaneously.

In this paper, we propose a novel regex synthesis framework based on the divide-and-conquer paradigm. Given a set of positive examples, we split examples into multiple parts by grouping similar substrings using a neural network and solve subproblems defined over the multiple parts in serial order. We train a neural network called the *NeuralSplitter* that embeds a set of positive examples into a fixed-size vector and produces the most probable split for each example in the set.

Our main contributions are as follows:

1. We propose a regex synthesis framework *SplitRegex* based on the divide-and-conquer paradigm, which has been employed in many sequential algorithms. As far as we are aware, this is the first attempt to synthesize regexes in the divide-and-conquer manner.
2. We design a neural network called the *NeuralSplitter*—a two-stage attention-based recurrent neural network trained to predict the most probable split of input positive strings.
3. We conduct exhaustive experiments on various benchmark datasets including random dataset and practical regex datasets for demonstrating the effectiveness of the the proposed framework. We show that our framework can be easily combined with the off-the-shelf regex synthesis algorithms with minor modifications.

Namely, our framework solves regex synthesis problem, which is PSPACE-complete, much faster when applied to any existing regex synthesis models, and thus boosts the overall performance.

2 Related Work

We provide a list of related works on regex synthesis problems based on various input types including positive and negative examples and natural language descriptions.

2.1 Regular Expression Inference from Examples

One approach to regular expression inference is by Angluin's (1987). Angluin suggests an algorithm L^* that infers an unknown DFA in teacher-student architecture. The teacher of L^* offers membership queries and equivalent queries on the target expression for the student. Note that this type of algorithms generally requires an infinite amount of examples to synthesize an FA.

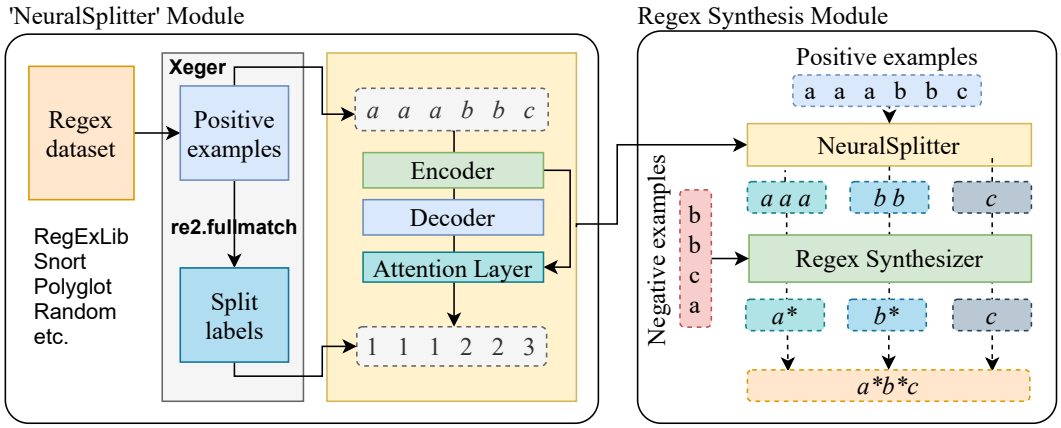

Figure 1: The overview of the proposed 'SplitRegex' regex synthesis framework.

Gold (1978) proposes a heuristic inference algorithm that inductively distinguishes random prefixes according to the acceptance followed by the given examples. The prefixes with the same acceptance form a single state in a DFA. The blue-fringe algorithm (Lang et al., 1998) and RPNI (Oncina & García, 1992) are similar to the Gold's algorithm.

AlphaRegex (Lee et al., 2016) exhaustively searches through all possible regular expressions until it identifies a solution regex that satisfies given positive and negative examples. A major limit of AlphaRegex is that its runtime is extremely due to the exponential increase of search space.

RegexGenerator++ (Bartoli et al., 2014; 2016) produces a regular expression using the genetic programming approach. The produced regular expression recognizes the surroundings (context) of the target sequence as well as the target itself. However, RegexGenerator++ entails a huge amount of computations due to the repetitive nature of genetic approach on each inference.

2.2 Regular Expression Synthesis from Natural Language Descriptions

Ranta (1998) studies rule-based techniques for the conversion between multi-languages and regular expressions. Kushman & Barzilay (2013) build a parsing model that translates a natural language sentence to a regular expression, and provides a dataset, which is the most popular benchmark dataset in recent research. Locascio et al. (2016) propose the Deep-Regex model based on Seq2Seq for generating regular expressions from NL descriptions together with 10,000 NL-RX pair data. However, due to the limitations of the standard Seq2Seq model, the Deep-Regex model can only generate regular expressions similar in shape to the training data. The SemRegex model (Zhong et al., 2018b) improves the Deep-Regex model by reinforcement learning based on the DFA-equivalence oracle, which determines if two regular expressions define the same language, as a reward function and their model can generate correct regular expressions even if they do not exactly look like answers. The SoftRegex model (Park et al., 2019) further improves SemRegex based on the fast regex equivalence test, which gives rise to a fast learning process.

2.3 Multi-modal Regular Expression Synthesis

Most multi-modal synthesis method uses natural language descriptions and positive/negative examples simultaneously to mitigate the ambiguity in natural languages. Chen et al. (2020) introduce hierarchical sketch for the semantics of natural description. Chen et al.'s implementation Regel first deduces the hierarchical structure representing the semantics of the natural language description. This semantic structure has some holes which the regular expression inference algorithm enhances a regular expression by predefined rules. Regel also prunes an infeasible expression using over- and under-approximation possible regular expressions, which replaces the holes with the most permissive and the most restrictive expression, respectively. Li et al. (2021) propose TransRegex framework that synthesizes a valid regular expression. TransRegex's NLP-based synthesizer translates the given

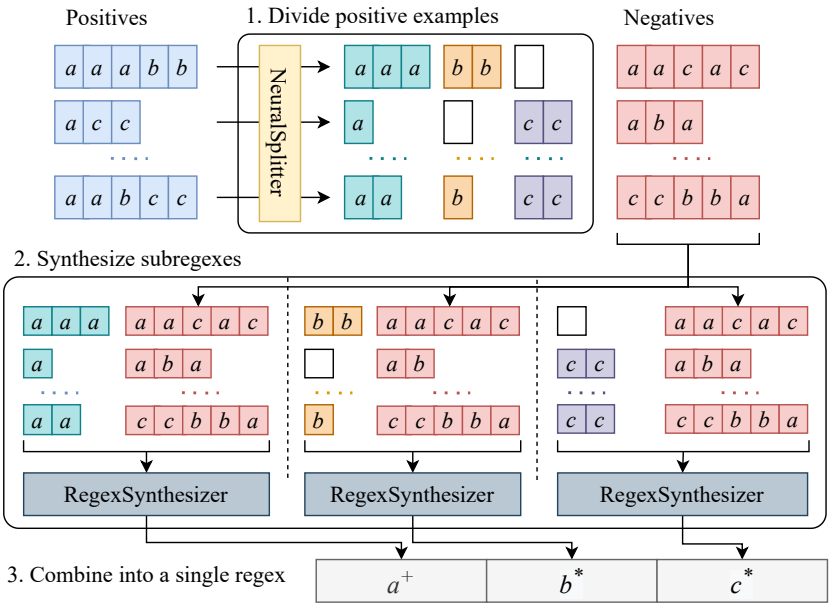

Figure 2: Overview of the proposed approach

description to a regular expression and fixes the expression by rules and RFixer (Pan et al., 2019) to increase matching score on the examples.

2.4 Repair-base Regular Expression Synthesis

Pan et al. (2019) design a heuristic algorithm RFixer that repairs a regular expression with given examples. RFixer improves the given regular expression by introducing or removing a hole and approximate the target expression with over- and under-approximation method until finds a regular expression with no holes that satisfies the given examples. Li et al. (2020) propose FlashRegex that generates a deterministic regular expression, which is a subclass of regular expressions. A SAT solver designs the basic FA structure after the given examples and FlashRegex iteratively shuffles the transitions in FA model at random to find the most suitable FA that satisfies the given examples.

3 Our Approach

Our approach involves the following procedure:

1. Split the input positive strings in the set P into multiple partitions $P_1, P_2, \ldots, P_S$, where S is the number of splits, using the NeuralSplitter model.
2. Run the regex synthesis engine for each P_i for $1 \leq i \leq S$ with a corresponding set N_i of negative strings. We will explain in more detail about how to use the set N of negative strings for final regex synthesis.
3. Concatenate the obtained subregexes $R_1, R_2 \ldots, R_S$ such that $P_i \subseteq L(R_i)$ to produce the final regex R.

Note that we cannot split negative examples just as we do for positive examples using the NeuralSplitter model as there may be another split (not produced by the NeuralSplitter) that results in negative strings recognized by the synthesized regex. For instance, consider a simple case when given $P = \{\mathsf{aabbcc}, \mathsf{abc}, \mathsf{bbcc}\}$ and $N = \{\mathsf{aabbccc}\}$. The NeuralSplitter may simply predict that substrings with the same repeated symbols should be grouped to form a subregex. Then, the first part has positive examples $P_1 = \{\mathsf{aa}, \mathsf{a}, \lambda\}$ and $N_1 = \{\mathsf{aa}\}$, the second part has $P_2 = \{\mathsf{bb}, \mathsf{b}, \mathsf{bb}\}$ and $N_2 = \{\mathsf{bb}\}$, and finally the last part has $P_3 = \{\mathsf{cc}, \mathsf{c}\}$ and $N_3 = \{\mathsf{ccc}\}$. It is possible that the regex synthesis algorithm generates $\mathsf{a}^*\mathsf{b}^*\mathsf{cc}^?$, which accepts the negative example.

We resolve this problem by introducing the concept of *prefix-conditioned subregex synthesis*. The basic idea of our approach is to first synthesize $n-1$ subregexes once the NeuralSplitter produces n splits $(P_1, P_2, \ldots, P_n)$ from P. Then, we concatenate the resulting $n-1$ regexes to be $R_1 R_2 \cdots R_{n-1}$ and use this regex to obtain the final regex from the last split sample P_n that satisfies all prefix subregexes as well as the negative samples. Namely, for each regex synthesis R_i, we only consider the corresponding split examples P_i from P. We use the negative samples to prevent for a regex synthesis algorithm to return a trivial regex such as the wildcard pattern regex $.^*$ (capturing zero or more any characters). One may consider to use the NeuralSplitter approach for the negative samples. Our empirical observation confirms that it causes substantially larger amount of additional computations, and the performance becomes worsened.

When there are common strings between the split positive examples P_i and the negative examples N, we remove the common strings from N. If P_i has only one example (singleton set), then we simply returns the example as R_i. The proposed method generates a regex R from a set P of positive examples and a set N of negative examples that satisfies the following conditions:

- $L(R_i) \subseteq P_i$ and $L(R_i) \cap (N \setminus P_i) = \emptyset$ for $1 \leq i \leq n-1$ (independent subregex synthesis).
- $L(R_1 R_2 \cdots R_n) \cap N = \emptyset$ (prefix-conditioned subregex synthesis).
- $L(R_i) = P_i$ if $|P_i| = 1$ for $1 \leq i \leq n$ (singleton subregex synthesis).

While the proposed method can generate a regex quickly by dividing a complex synthesis problem into several easier problems, one limit is that it cannot generate the wildcard pattern regex, $.^*$, as a subregex R_i due to the negative examples. Because the wildcard pattern is common and handy in practice to represent complicated sequences, we treat the symbols generated from the wildcard pattern differently compared with the rest of the symbols—we assign a special label 0 for the symbols in the data preprocessing step. For instance, if a string abbccdd is generated from $\mathsf{a}^*\mathsf{b}^*\mathsf{c}^*\mathsf{d}^*$, then the label is 1223344. However, if abbccdd is generated from $\mathsf{a}^*.^*$, then the label is 1000000. When reading these special symbol 0's, our framework immediately generates wildcard patterns without going through the prefix-conditioned subregex synthesis. A detailed procedure for the data prepossessing is in Section 3.1.

3.1 Data Preprocessing

From the dataset, we exclude or modify regexes with features listed below:

- Backreference: we do not consider backreference for regex synthesis as some baseline approaches such as AlphaRegex and Blue-Fringe algorithm do not support regex features that capture non-regular languages.
- Lookahead/lookbehind: while regexes with these features still only capture regular languages, we do not consider these features as they require the regex engine to perform backtracking that causes catastrophic runtime in some extreme cases.
- Quantifier with exact count '$\{\mathsf{n}\}$' or range '$\{\mathsf{n},\mathsf{m}\}$': we replace numerical quantifiers with '$*$' to 1) reduce the search space and 2) control the length of randomly generated strings.

For each regex, we randomly generate 20 positive and negative examples of length up to certain length (10 for random dataset and 15 for the other datasets). In order to generate positive examples, we utilize a Python library called the Xeger[1]. For generating negative examples, we first generate positive examples and randomly substitute symbols with other symbols to make the examples outside the language represented by the regex. We find this is effective compared to the complete random generation of negative examples.

In order to guarantee that every regular expression has 20 positive and negative examples, we exclude regular expressions that do not generate more than 20 positive strings (e.g., regexes without quantifiers such as $*$ and $+$). From 20 positive and negative strings, we use 10 positive/negative strings for generating regexes and use the remaining strings for measuring the semantic accuracy of synthesized regexes by membership queries.

[1] https://pypi.org/project/xeger/

We generate output labels for positive examples as follows. We use Python library pyre2, which is a Python wrapper for Google's RE2 regex library, to perform matchings of positive examples against R. More specifically, we run the fullmatch method of pyre2 to find subregexes from the whole regex that match corresponding substrings from given positive strings.

For instance, when we are given $P = \{\mathsf{aabbccabca}, \mathsf{abcbbc}, \mathsf{bbbcabb}, \mathsf{aabbbc}\}$ and $R = \mathsf{a}^*\mathsf{b}^*\mathsf{c}^*.^*$, we obtain ground truth labels for the four positive examples in P as follows: $L_P = \{1122330000, 123000, 2223000, 112223\}$.

3.2 Neural Network Architecture

We introduce a neural network called the *NeuralSplitter* that is trained to split positive examples generated from each regex. For a set $P = \{p_1, p_2, \ldots, p_n\}$ of positive examples, where $p_i \in \Sigma^*$ for $1 \leq i \leq n$, we embed each string p_i with an RNN encoder into d dimensional space as follows:

$$h_i = \mathrm{RNN}_{\mathsf{enc1}}(p_i),$$

where $h_i \in \mathbb{R}^d$. Then, we embed the set P of positive examples with the second RNN encoder $\mathrm{RNN}_{\mathsf{enc2}}$ as follows:

$$h_P = \mathrm{RNN}_{\mathsf{enc2}}(h_1, h_2, \ldots, h_n),$$

where $h_P \in \mathbb{R}^d$.

From the hidden vectors h_i for $1 \leq i \leq n$ and h_P, the RNN decoder is trained to produce the ground truth label for each positive string p_i as follows:

$$o_1, o_2, \ldots, o_{l_i} = \mathrm{RNN}_{\mathsf{dec}}(W_{\mathsf{dec}}^\top \cdot [h_i; h_P]),$$

where $l_i = |p_i|$, $o_j \in \mathbb{R}^d$ for $1 \leq j \leq l_i$, $W_{\mathsf{dec}} \in \mathbb{R}^{2d \times d}$.

Finally, we retrieve the predicted label for each positive example p_i as follows:

$$y_j = \mathrm{softmax}(W_{\mathsf{output}}^\top \cdot o_j),$$

where $y_j \in \mathbb{R}^{|V|}$, $W_{\mathsf{output}} \in \mathbb{R}^{d \times |V|}$, and V is the output vocab. Note that y_j is the predicted probability distribution for the label of jth symbol of the positive example p_i.

During the decoding process, the RNN decoder exploits the attention mechanism to better point out the relevant parts from the entire set of positive strings. Since the split label for a positive string relies on the information from all positive strings, we implement two-step attention mechanism that enables our model to attend both in set level and example level. For instance, when we have two strings aabbbc and aabbcc, the most probable labels for these two strings are 112223 and 112233. However, if the second string is caabbcc, where only the first symbol c is appended in front of aabbcc, then the most probable labels for two strings become 223334 and 1223344. In this reason we need to use the information of all given positive strings to correctly predict the output labels.

We use GRU cells for RNN encoder and decoder of NeuralSplitter. Cross entropy loss is used for training neural network.

4 Experimental Setup

We describe the experimental setup for evaluation.

Datasets. We use the following benchmarks to measure the performance of the proposed approach compared to the previous approaches.

- Random dataset: consists of 1,000,000 randomly generated regexes (784,682 unique regexes) over various alphabet sizes 2, 4, 6, 8 and 10.
- RegExLib[2]: consists of regexes collected from an online regex library where 4,149 regexes are indexed from 2,818 contributors to help regex users write their own regexes more easily.

[2] https://www.regexlib.com/

- Snort[3]: an open source intrusion detection/prevention system, contains more than 1,200 regexes in its rule set for compactly describing malicious network traffic.
- Polyglot Regex Corpus (Davis et al., 2019): consists of 537,806 regexes extracted from 193,574 software projects written in eight popular programming languages. Introduced for the research on regex portability problem.

For training the NeuralSplitter and evaluating our framework on practical regex benchmarks including RegExLib, Snort and Polyglot Regex Corpus, we first generate 20 random positive and negative examples for each regex and split the data into training, validation and test set with the ratio of 8:1:1. Then, we collect the training and validation splits for three benchmarks for training the NeuralSplitter model, and use each test set for evaluating the regex synthesis performance on each benchmark.

Hyperparameters. We use two-layer bidirectional GRU with 256 hidden units unless explicitly mentioned otherwise. The dimension of input token embedding is four. The Adam optimizer is used with batch size 512 and learning rate of 0.001 and weight decay of 1e-6.

Baselines. We take the following baseline methods for regex synthesis from examples.

- AlphaRegex (Lee et al., 2016; Kim et al., 2021): a best-first search based regex enumeration algorithm. We use our own Python implementation of AlphaRegex algorithm, which is originally written in OCaml[4] together with the recent search space reduction techniques by Kim et al. (2021).
- RegexGenerator++ (Bartoli et al., 2014; 2016): a regex inference algorithm based on genetic programming. Due to the nature of genetic inference procedure, this is the slowest among the baselines.
- Blue-Fringe algorithm (Lang et al., 1998): a classical state-merging DFA induction algorithm. We choose this algorithm to show the generality of our framework on classical synthesizers such as L^* (Angluin, 1987) or RPNI (Oncina & García, 1992).

Note that we also use these baselines as off-the-shelf regex synthesizers for our SplitRegex framework to evaluate the performance boost gained by our framework.

Metrics. We evaluate the performance of the proposed approach in two main aspects: 1) the time efficiency of synthesis algorithm and 2) the accuracy of synthesized regexes. For evaluating the time efficiency, we set a time budget, specified by a *timeout* value. We set the timeout value to 3 (seconds) in our experiments. A regex synthesis is counted as a *success* if an algorithm finds a regex satisfying given positive and negative examples within the specified timeout. We measure the *success rate* of regex synthesis by counting the number of successes in n regex synthesis trials where n is the number of target regexes in test set.

Meanwhile, we also need to measure the accuracy of synthesized regexes as any algorithm may generate a trivial solution that only recognizes positive examples. However, we cannot simply count the number of cases where the synthesized regex is 'literally' equivalent to the target regex as two different regexes can capture the same set of strings as there are infinitely many semantically equivalent regexes. Hence, we need to introduce a criterion that considers how two regexes are semantically similar. While it is well-known that it is possible to algorithmically decide whether two regexes represent the same language, it is also well-known that the decision process is PSPACE-complete (Stockmeyer & Meyer, 1973) and therefore probably intractable.

Therefore, we define a metric called the *semantic regex accuracy* that utilizes randomly generated positive and negative examples that are not used for regex synthesis. Remark that the semantic regex accuracy has been already employed in a previous work for evaluating the generated regexes (Li et al., 2021). Given a pair (P, N) of positive and negative sets reserved for evaluation, we define the semantic regex accuracy as follows:

$$\text{sem_accuracy}(R, P, N) = \frac{|T_P| + |T_N| - |F_P| - |F_N|}{|P| + |N|} \times 100,$$

[3] `https://www.snort.org/`

[4] `https://github.com/kupl/AlphaRegexPublic`

Table 1: Regex synthesis performance (success rate and semantic regex accuracy) of our model and baselines over four benchmark regex datasets. 'AR' and 'BF' are abbreviations for AlphaRegex and Blue-Fringe algorithms, respectively. 'Random' dataset consists of regexes over alphabet of size ten.

Method	**Random**		**RegExLib**		**Snort**		**Polyglot Corpus**	
	Succ.	**Acc.**	**Succ.**	**Acc.**	**Succ.**	**Acc.**	**Succ.**	**Acc.**
AlphaRegex	48.10	44.94	16.30	15.34	15.55	13.50	47.80	52.26
RegexGenerator++	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Blue-Fringe	**99.80**	0.00	**97.30**	0.86	**98.70**	0.50	**95.70**	4.44
SplitRegex (+AR)	63.70	**60.52**	20.30	**18.10**	57.77	**56.88**	56.00	**57.01**
SplitRegex (+RG)	0.00	0.00	1.00	1.00	3.00	3.00	20.00	20.00
SplitRegex (+BF)	95.80	36.48	76.30	11.10	95.00	16.56	90.00	12.94

Table 2: Comparisons of different regex synthesis method using split positive examples.

Synthesis Strategy	**On Random Regexes**		**On Practical Regexes**	
	Success Rate	**Accuracy**	**Success Rate**	**Accuracy**
Baseline (AlphaRegex)	47.57	44.83	12.90	13.08
Prefix-conditioned synthesis	63.90	60.50	37.60	36.94
Independent parallel synthesis	42.00	40.40	54.50	54.34
Independent sequential synthesis	**64.60**	**61.68**	**55.84**	**55.20**

where $T_P = \{w \in L(R) \mid w \in P\}$, $T_N = \{w \notin L(R) \mid w \in N\}$, $F_P = \{w \notin L(R) \mid w \in P\}$ and $F_N = \{w \in L(R) \mid w \in N\}$.

Intuitively, the semantic regex accuracy is higher if the synthesized regex recognizes more strings recognizable by the target regex and rejects more strings not recognizable by the target regex.

5 Experimental Results and Analysis

We demonstrate the experimental results to show the effectiveness of the proposed approach compared to baselines and analyze the results.

5.1 Main Results

Table 1 shows the main result of our paper. We evaluate our approach with the baselines in terms of both success rate and semantic accuracy of synthesized regexes using reserved examples.

It is readily seen that the proposed framework substantially improves both the success rate and semantic accuracy in most benchmarks except that Blue-Fringe algorithm achieves 99.80% success rate for random dataset. While the original Blue-Fringe algorithm returns solutions satisfying the given examples within the timeout (3 seconds) due to its nature, the semantic accuracy of the produced regexes is very low as they accept most of reserved negative examples. However, when our SplitRegex framework is combined with Blue-Fringe, the semantic accuracy is improved even though it fails to return solutions for some cases due to additional overhead caused by neural network computations.

5.2 Detailed Analysis

Alphabet size. Figure 3 shows how the SplitRegex framework affects the regex synthesis performance of AlphaRegex and Blue-Fringe algorithms on different alphabet sizes. While the success rate decreases as the alphabet size increases for AlphaRegex, SplitRegex effectively improves the success rate as the alphabet size increases. Moreover, the semantic accuracy of synthesized regexes is also improved when the SplitRegex is applied to both AlphaRegex and Blue-Fringe.

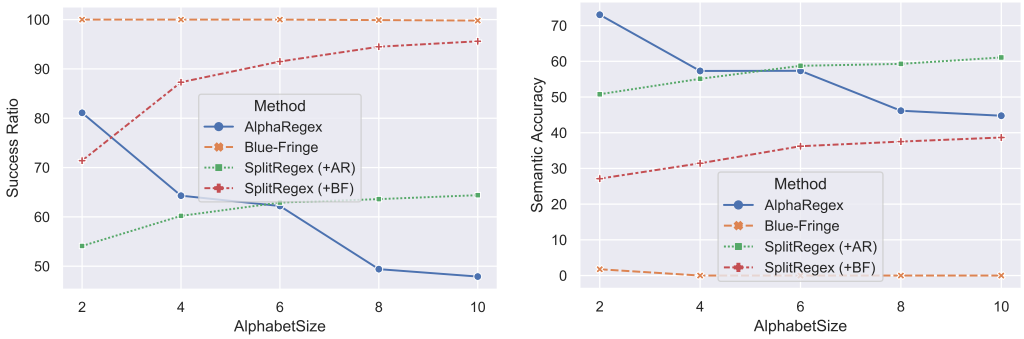

Figure 3: Results on random regular expression dataset with different alphabet sizes.

Table 3: Examples of neural example splitting for two benchmark datasets.

Dataset	**Input String**	**Split Result**				
RegExLib	uname=&pword=9E8	uname=		&pword=	9E8	
	uname=!&pword=	uname=	!	&pword=		
	uname=Hj%}v&pword=‘Af7=	uname=	Hj%}v	&pword=	‘Af7=	
	uname=_pp}C&pword=Ht	uname=	_pp}C	&pword=	Ht	
	uname=)#&pword=@k	uname=	)#	&pword=	@k	
Snort	/images!php?t=86153/U	/images	!	php?t=	86153	/U
	/images-php?t=296/U	/images	-	php?t=	296	/U
	/images4php?t=5213/U	/images	4	php?t=	5213	/U
	/images{php?t=15/U	/images	{	php?t=	15	/U
	/imagesQphp?t=165/U	/images	Q	php?t=	165	/U

Subregex synthesis strategies. We conduct additional experiments to verify whether the ‘prefix-conditioned synthesis’ strategy can be utilized for every subsequent subregexes while we utilize only for synthesizing the last subregex. Moreover, we also test whether the first $n-1$ subregexes (when the split size is n) can be synthesized in parallel if the prefix-conditioned synthesis is not used. Table 7 demonstrates the experimental results on random regexes and practical regexes from RexExLib, Snort and Polyglot corpus datasets. The results imply that the simplest approach (independent sequential synthesis except the final subregex) achieves the best performance compared to the parallel approach and prefix-conditioned synthesis approach.

Successful examples. Table 3 demonstrates several successful examples of neural example splitting. For instance, positive examples in the ‘Snort’ dataset is generated from a regex ‘/images.*php?t=.*/U’. We can see that our NeuralSplitter model successfully splits positive examples into five parts where similar strings are grouped together. As a result, SplitRegex produces the following regex for the examples which is quite similar to the target regex: ‘/images.*php?t=[0-9]*/U’.

6 Conclusions and Future Work

In this paper, we have proposed a regex synthesis framework called the SplitRegex that synthesizes a regex from given positive and negative examples based on the divide-and-conquer paradigm. Given a set of positive examples, we predict the most probable splits for strings such that substrings in the same partition exhibit similar patterns using a neural network called the NeuralSplitter. Then, we synthesize subregexes for all partitions and derive the final regex satisfying the input examples. Our experimental results show that the proposed scheme is very effective in reducing the time complexity of regex synthesis and actually yields more accurate regexes than the previous approaches.

REFERENCES

Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. Scaling enumerative program synthesis via divide and conquer. In Axel Legay and Tiziana Margaria (eds.), *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017*, volume 10205 of *Lecture Notes in Computer Science*, pp. 319–336, 2017.

Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In *3rd International Conference on Learning Representation, ICLR 2015*, 2015.

A. Bartoli, G. Davanzo, A. De Lorenzo, E. Medvet, and E. Sorio. Automatic synthesis of regular expressions from examples. *Computer*, 47(12):72–80, dec 2014. ISSN 1558-0814. doi: 10.1109/MC.2014.344.

Alberto Bartoli, Andrea De Lorenzo, Eric Medvet, and Fabiano Tarlao. Inference of regular expressions for text extraction from examples. *IEEE Trans. on Knowl. and Data Eng.*, 28(5):1217—-1230, May 2016.

Qiaochu Chen, Xinyu Wang, Xi Ye, Greg Durrett, and Isil Dillig. Multi-modal synthesis of regular expressions. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 487–502, 2020.

James C. Davis, Louis G. Michael IV, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. Why aren't regular expressions a lingua franca? an empirical study on the re-use and portability of regular expressions. In Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo (eds.), *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019*, pp. 443–454. ACM, 2019.

Azadeh Farzan and Victor Nicolet. Counterexample-guided partial bounding for recursive function synthesis. In Alexandra Silva and K. Rustan M. Leino (eds.), *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I*, volume 12759 of *Lecture Notes in Computer Science*, pp. 832–855. Springer, 2021a.

Azadeh Farzan and Victor Nicolet. Phased synthesis of divide and conquer programs. In Stephen N. Freund and Eran Yahav (eds.), *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 20211*, pp. 974–986. ACM, 2021b.

E. Mark Gold. Complexity of automaton identification from given data. *Information and Control*, 37 (3):302–320, 1978.

Su-Hyeon Kim, Hyeonseung Im, and Sang-Ki Ko. Efficient enumeration of regular expressions for faster regular expression synthesis. In Sebastian Maneth (ed.), *Implementation and Application of Automata*, pp. 65–76. Springer International Publishing, 2021.

Nate Kushman and Regina Barzilay. Using semantic unification to generate regular expressions from natural language. In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 826–836, 2013.

Kevin J. Lang, Barak A. Pearlmutter, and Rodney A. Price. Results of the abbadingo one DFA learning competition and a new evidence-driven state merging algorithm. In Vasant G. Honavar and Giora Slutzki (eds.), *Grammatical Inference, 4th International Colloquium, ICGI-98, Ames, Iowa, USA, July 12-14, 1998, Proceedings*, volume 1433 of *Lecture Notes in Computer Science*, pp. 1–12. Springer, 1998.

Mina Lee, Sunbeom So, and Hakjoo Oh. Synthesizing regular expressions from examples for introductory automata assignments. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE 2016, pp. 70—-80, 2016.

Yeting Li, Zhiwu Xu, Jialun Cao, Haiming Chen, Tingjian Ge, Shing-Chi Cheung, and Haoren Zhao. Flashregex: Deducing anti-redos regexes from examples. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ASE '20, pp. 659—-671, 2020.

Yeting Li, Shuaimin Li, Zhiwu Xu, Jialun Cao, Zixuan Chen, Yun Hu, Haiming Chen, and Shing-Chi Cheung. TransRegex: multi-modal regular expression synthesis by generate-and-repair. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*, pp. 1210–1222. IEEE, 2021.

Nicholas Locascio, Karthik Narasimhan, Eduardo DeLeon, Nate Kushman, and Regina Barzilay. Neural generation of regular expressions from natural language with minimal domain knowledge. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pp. 1918–1923, 2016.

J. Oncina and P. García. *Inferring Regular Languages In Polynomial Updated Time*, pp. 49–61. 1992.

Rong Pan, Qinheping Hu, Gaowei Xu, and Loris D'Antoni. Automatic repair of regular expressions. *Proc. ACM Program. Lang.*, 3(OOPSLA), 2019.

Jun-U Park, Sang-Ki Ko, Marco Cognetta, and Yo-Sub Han. SoftRegex: Generating regex from natural language descriptions using softened regex equivalence. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pp. 6425–6431, 2019.

Aarne Ranta. A multilingual natural-language interface to regular expressions. In *Proceedings of the International Workshop on Finite State Methods in Natural Language Processing*, pp. 79–90, 1998.

Larry J Stockmeyer and Albert R Meyer. Word problems requiring exponential time (preliminary report). In *Proceedings of the fifth annual ACM symposium on Theory of computing*, pp. 1–9, 1973.

Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pp. 3104–3112, 2014.

Zexuan Zhong, Jiaqi Guo, Wei Yand, Tao Xie, Jian-Guang Lou, Ting Liu, and Dongmei Zhang. Generating regular expressions from natural language specifications: Are we there yet? In *Workshops at the 32nd AAAI Conference on Artificial Intelligence*, pp. 791–794, 2018a.

Zexuan Zhong, Jiaqi Guo, Wei Yang, Jian Peng, Tao Xie, Jian-Guang Lou, Ting Liu, and Dongmei Zhang. Semregex: A semantics-based approach for generating regular expressions from natural language specifications. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pp. 1608–1618, 2018b.

APPENDIX

A BACKGROUND

Here we provide some background knowledge on the task of regular expression synthesis/inference and basic neural network architectures used in our model.

Regular expression inference/synthesis. Let Σ be a finite alphabet and Σ^* be the set of all strings over the alphabet Σ. A *regular expression* (regex) over Σ is $a \in \Sigma$, or is obtained by applying the following rules finitely many times. For regexes R_1 and R_2, the union $R_1 + R_2$, the concatenation $R_1 \cdot R_2$, the star R_1^*, and the question $R_1^?$ are also regexes. Note that $L(R_1^?)$ is defined as $L(R_1) \cup \{\epsilon\}$.

Given a set of positive and negative strings, we consider the problem of synthesizing a concise regex that is consistent with the given strings. The examples are given by a pair (P, N) of two sets of strings, where $P \subseteq \Sigma^*$ is a set of *positive strings* and $N \subseteq \Sigma^*$ is a set of *negative strings*.

Then, our goal is to find a regex R that accepts all positive strings in P while rejecting all negative strings in N. Formally, R satisfies the following condition:

$$P \subseteq L(R) \text{ and } L(R) \cap N = \emptyset.$$

Recurrent network networks. An RNN is described by a function that takes an input x_t and a hidden state h_{t-1} at time step $t-1$ and returns a hidden state h_t at time step t as follows: $h_t = \mathrm{RNN}(x_t, h_{t-1})$.

Given an input sentence $\mathbf{x} = (x_1, x_2, \ldots, x_n)$, an encoder $\mathrm{RNN}_{\mathsf{enc}}$ plays its role as follows: $h_t = \mathrm{RNN}_{\mathsf{enc}}(x_t, h_{t-1})$ where $h_t \in \mathbb{R}^d$ is a hidden state at time t. After processing the whole input sentence, the encoder generates a fixed-size context vector that represents the sequence as follows: $c = q(h_1, h_2, \ldots, h_n)$.

Usually, q simply returns the last hidden state h_n in one of the original sequence to sequence paper by Sutskever et al. (2014). Since the initial hidden state h_0 is usually randomly initialized, we simply define RNN as a function that takes sequence $\mathbf{x}$ and produces a vector c as follows: $c = \mathrm{RNN}(\mathbf{x})$.

Now suppose that $\mathbf{y} = (y_1, y_2, \ldots, y_m)$ is an output sentence that corresponds to the input sentence $\mathbf{x}$ in training set. Then, the decoder RNN is trained to predict the next word conditioned on all the previously predicted words and the context vector from the encoder RNN. In other words, the decoder computes a probability of the translation $\mathbf{y}$ by decomposing the joint probability into the ordered conditional probabilities as follows:

$$p(\mathbf{y}) = \prod_{i=1}^{m} p(y_i | \{y_1, y_2, \ldots, y_{i-1}\}, c).$$

Now our decoder $\mathrm{RNN}_{\mathsf{dec}}$ computes each conditional probability as follows:

$$p(y_i | y_1, y_2, \ldots, y_{i-1}, c) = \mathrm{softmax}(W^\top \cdot s_i),$$

where $s_i \in \mathbb{R}^d$ is the hidden state of decoder RNN at time i and $W \in \mathbb{R}^{d \times |V|}$ is a linear transformation that outputs a vocabulary-sized vector. Note that the hidden state s_i is computed by

$$s_i = \mathrm{RNN}_{\mathsf{dec}}(y_{i-1}, s_{i-1}, c),$$

where y_{i-1} is the previously predicted word, s_{i-1} is the last hidden state of decoder RNN, and c is the context vector computed from encoder RNN.

Attention mechanism. In order to resolve the long-range dependency problem in the sequence-to-sequence model, Bahdanau et al. (2015) defined each conditional probability at time i depending on a dynamically computed context vector c_i as follows:

$$p(y_i | y_1, y_2, \ldots, y_{i-1}, \mathbf{x}) = \mathrm{softmax}(g(s_i)),$$

where s_i is the hidden state of the decoder RNN at time i computed by $s_i = \mathrm{RNN}_{\mathsf{dec}}(y_{i-1}, s_{i-1}, c_i)$.

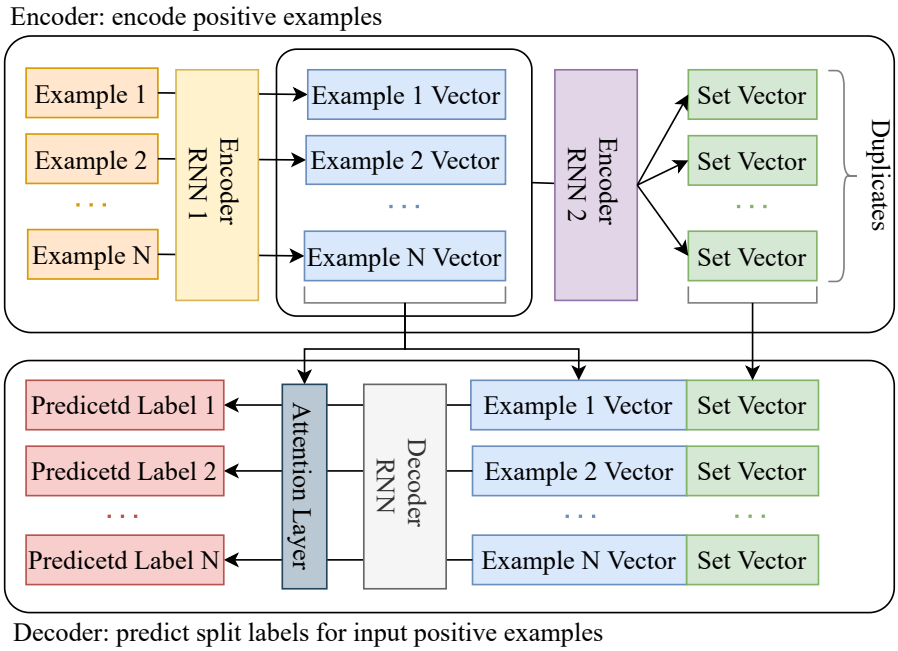

Figure 4: Neural network architecture of the 'NeuralSplitter' model.

The context vector c_i is computed as a weighted sum of the hidden states from encoder: $c_i = \sum_{j=1}^{n} \alpha_{ij} s_j$, where

$$\alpha_{ij} = \frac{\exp(\text{score}(s_i, h_j))}{\sum_{k=1}^{n} \exp(\text{score}(s_k, h_j))}.$$

Here the function 'score' is called an *alignment function* that computes how well the two hidden states from the encoder and the decoder, respectively, match. For example, $\text{score}(s_i, h_j)$, where s_i is the hidden state of the encoder at time i and h_j is the hidden state of the decoder at time j implies the probability of aligning the part of the input sentence around position i and the part of the output sentence around position j.

B Neural Network Architecture

Table 4 depicts the neural network architecture of the 'NeuralSplitter' model.

C Experimental Details

C.1 Statistics of Datasets

We refer the readers to Table 4 for the statistics of benchmark datasets. For random dataset, we randomly generate in total 1,000,000 regexes independently where there are 200,000 regexes for each alphabet size in $\{2, 4, 6, 8, 10\}$. The number of unique regexes is 784,682 as we allow duplicates in random generation.

Environment. We conduct experiments on a server equipped with Intel Core i7-8700K processor with 6 cores, 48GB DDR4 memroy, and NVIDIA GTX 1080 Ti GPU. We use the neural network framework PyTorch for experiments.

Table 4: Statistics of four benchmark datasets. Numbers in parentheses are the numbers of regexes in original datasets and in front of the parentheses are the numbers of regexes used in our experiments.

Dataset	**# of Regexes**	**Alphabet Size**	**Max Length**
Random	784,682	$\{2, 4, 6, 8, 10\}$	81
RegExLib	1,422 (3,072)	128	767
Snort	375 (1,254)	128	203
Polyglot Corpus	199,350 (537,800)	128	1,903

Table 5: Split performance (%) of the NeuralSplitter under different settings of hyperparameters.

Dataset	**RNN Cell**	**Dimension**	**# of Layers**	**String Accuracy**	**Set Accuracy**
Random	GRU	64	1	54.10	35.04
			2	55.34	37.04
		128	1	57.41	38.03
			2	57.41	38.03
		256	2	57.41	38.24
		384	2	57.21	38.25
	LSTM	256	2	**57.93**	**38.34**
Practical	GRU	256	2	**76.78**	**68.20**

D Additional Results

D.1 Performance of 'Neural Example Splitting'

The performance of the NeuralSplitter model under different settings of hyperparameters and RNN cell types is provided in Table 5. We define two metrics, 'string accuracy' and 'set accuracy', for evaluating the performance of neural example splitting. String accuracy is the ratio of correctly labeled examples from ten positive examples for each test sample. Set accuracy means the ratio of correctly labeled sets as a whole. If the NeuralSplitter correctly splits ten positive examples for a test sample, then it is regarded as 'correctly labeled'. It is notable that the NeuralSplitter performs better on practical dataset which is apparently more complex than random dataset.

While we find that LSTM cell performs slightly better than GRU cell when the other hyperparameters are equivalent, we decide to use GRU instead of LSTM as GRU is known to execute faster than LSTM. We also find that the performance of the NeuralSplitter is higher as we increase the number of hidden units and the number of layers until 256 and 2, respectively. We decide not to increase the number of parameters further as the computation efficiency is also a very important metric in the task of regex synthesis.

D.2 Performance Comparison with Different Timeout Values

Table 6 shows how the regex synthesis performance changes as the timeout value changes. We perform regex synthesis with various timeout values 1, 2 and 3 seconds and measure the success ratio and semantic accuracy of the proposed baselines including our approach. We do not include RegexGenerator++ for comparison as it fails to synthesize regexes in most cases even with timeout value of 3 seconds. The result shows that Blue-Fringe algorithm succeeds in regex synthesis within 1 second in most cases as the success rate and semantic accuracy does not increase with higher timeout values.

On the other hand, AlphaRegex with and without our SplitRegex framework exhibits better regex synthesis success rate and semantic accuracy as timeout value increases.

Table 6: Regex synthesis performance (success rate and semantic regx accuracy) of our model and baselines over four benchmark regex datasets under different timeout values. 'AR' and 'BF' are abbreviations for AlphaRegex and Blue-Fringe algorithms, respectively. 'Random' dataset consists of regular expressions over alphabet of size ten.

Method	**Time.**	**Random**		**RegExLib**		**Snort**		**Polyglot**	
		Succ.	**Acc.**	**Succ.**	**Acc.**	**Succ.**	**Acc.**	**Succ.**	**Acc.**
AlphaRegex	1	38.90	36.48	12.30	11.76	5.30	5.10	39.33	38.67
Blue-Fringe		**99.80**	0.00	**97.30**	0.86	**98.67**	0.59	**95.67**	4.47
SplitRegex (+AR)		55.90	**53.30**	19.00	**18.36**	58.30	**56.96**	57.33	**56.73**
SplitRegex (+BF)		95.80	36.50	71.30	11.10	94.67	16.57	89.00	12.96
AlphaRegex	2	44.30	41.50	15.30	14.36	6.70	6.36	44.33	43.23
Blue-Fringe		**99.80**	0.00	**97.33**	0.83	**98.67**	0.59	**95.67**	4.47
SplitRegex (+AR)		61.90	**58.84**	19.67	**18.89**	59.30	**57.84**	58.33	**57.67**
SplitRegex (+BF)		95.80	36.50	76.33	11.07	95.00	16.56	90.00	12.94
AlphaRegex	3	48.00	44.86	16.30	15.34	14.00	13.50	47.00	45.86
Blue-Fringe		**99.80**	0.00	**97.33**	0.83	**98.67**	0.59	**95.67**	4.47
SplitRegex (+AR)		63.70	**60.52**	20.30	**19.46**	59.30	**57.84**	59.00	**58.34**
SplitRegex (+BF)		95.80	36.50	76.33	11.07	95.00	16.56	90.00	12.94

Table 7: Comparisons of regex synthesis performance of AlphaRegex algorithm with and without SplitRegex framework on practical regex datasets.

Method	**Success Ratio**	**Win Ratio**	**Runtime**	**Runtime for Success Cases**
AlphaRegex	25.77	20.82	2.4344	0.1250
SplitRegex (+AR)	**46.10**	**79.18**	**1.6622**	**0.0088**

D.3 SPLIT VS NON-SPLIT

We evaluate the performance of our SplitRegex framework by comparing the results of regex synthesis model with and without the SplitRegex framework for each test case. First, we present the number of test cases that are exclusively solved by each method. 'Win Ratio' means the ratio of test cases where each method performs better (synthesizes regex faster). We do not count the cases where both methods fail to synthesize regexes when calculating 'Win Ratio (%)'.

We also estimate the average amount of time taken for regex synthesis in two different cases. First, 'Runtime (seconds)' is the average amount of time taken for synthesizing regexes in all test cases. If the method fails to synthesize a regex, then we consider it as '3 seconds' which is the timeout value used for this experiment. Second, 'Runtime for Success Cases (seconds)' is the average amount of time taken for synthesizing regexes in cases where both methods succeeds in synthesizing regexes.

Experimental results on practical regex datasets (RegExLib, Snort and Polyglot Corpus dataset) are provided in Table 7. Results show that SplitRegex sufficiently improves the success ratio and reduces runtime complexity taken for regex synthesis in almost 80% of test cases.