Visualizing the addition of missing words to regular expressions

Thomas Rebele[1], Katerina Tzompanaki[2], Fabian Suchanek[1]

[1] Télécom ParisTech, 75013 Paris, France
[2] ETIS / ENSEA, University of Cergy-Pontoise, CNRS, 95000, Cergy-Pontoise, France

Abstract. Regular expressions are used in many information extraction systems like YAGO, DBpedia, Gate and SystemT. However, they sometimes do not match what their creator wanted to find. We investigate how missing words can be added automatically to a regular expression by creating disjunctions at the appropriate positions. Our demo visualizes the steps that our algorithm employs to repair the regular expression.

1 Introduction

Regular expressions (regexes) are a popular tool in information extraction. While it is possible to learn regexes automatically from positive and negative examples, many projects craft their regexes manually. This applies, e.g., to Semantic Web projects such as YAGO and DBpedia, but also to systems such as Gate and SystemT. One of the reasons may be that the regexes contain human expertise that goes beyond the information contained in training data.

One of the challenges in such contexts is to adapt a regex so that it matches a word that was previously missed. Consider, e.g., the simple regex `[A-Za-z]+ \d\d, \d\d|\d\d/\d\d/\d\d\d\d`. It covers dates that spell out the month, and dates that consist of digits with a dash. However, it will not match "1/8/1935". One way to solve this is obviously to just add this word as a disjunction – but that would not generalize to other, similar words. To repair the regex properly, we would first have to identify where the word finds its largest approximate match in the regex (on the right hand side of the disjunction). Then, we have to perform a minimal surgical operation to make two digits optional. Finally, we could factor out the 4 digits of the year. In this paper, we present our work on an algorithm that does all of this automatically. In the example, we obtain `([A-Za-z]+ \d\d, |(\d{1,2}/){2})\d{4}`. Our demo allows the user to visualize these changes.

Related work. Several approaches [3, 2, 8] learn a regex from scratch. Our approach, in contrast, modifies a given regex. Other approaches remove false positives [4], while we aim to add false negatives. Again others [6] rely on user feedback, while we aim at a fully automated solution. Other work [1] generalizes individual characters, while we aim to change the *structure* of the regex.

2 Algorithm

Each regex r is associated to a syntax tree, whose leaf nodes are characters or character classes. We say that a leaf node is *to the left* of another leaf node, if the latter can be reached from the former in the Glushkov automaton of r. We write $L(r)$ for the language of r. Given a regex r and a word w, our goal is to construct a regex r' such that $w \in L(r') \wedge L(r) \subseteq L(r')$.

We first find a maximal matching of w to r, i.e., a partial function from the positions of w to the leaf nodes of r. For this purpose, we first unfold the syntactic sugar of the regex, so that it contains only Kleene stars, disjunctions, and concatenations. Then, we use Meyer's approximate matching algorithm [7]. Like the well-known Wagner-Fischer algorithm for the edit distance, Meyer's algorithm uses a matrix. The rows correspond to the characters of the word, and the columns correspond to leaf nodes of the regex (left to right). This gives us a mapping from positions in the string to leaf nodes of the regex.

Algorithm 1 Fix regex

INPUT: regex r, word w, matching m (a function from positions in w to leafs of r)
OUTPUT: modified regex r

1: **for** $i \in dom(m)$ in increasing order **do**
2: j $\leftarrow argmin_j \;\; j > i \wedge m(j)$ is defined
3: $n \leftarrow$ lowest common ancestor of $m(i)$ and $m(j)$ in the syntax tree of r
4: **if** $m(j)$ left of $m(i)$ in r **then**
5: $k \leftarrow \text{argmin}_k \quad k > j \wedge m(k)$ right of $m(i)$
6: **if** k exists **then**
7: Insert $(w_{i+1}...w_{k-1})$? below n between $m(i)$ and $m(k)$
8: $j \leftarrow k$
9: **else**
10: $m(j) \leftarrow$ undefined
11: **end if**
12: **continue**
13: **end if**
14: **for** node n' on the path from $m(i)$ to n in the syntax tree of r **do**
15: **if** n' is concatenation **then** make children of n' to the right optional
16: **end for**
17: **for** node n' on the path from $m(j)$ to n in the syntax tree of r **do**
18: **if** n' is concatenation **then** make children of n' to the left optional
19: **end for**
20: $c_1, ..., c_k \leftarrow$ children of n between $m(i)$ and $m(j)$
21: Replace $c_1, ..., c_k$ by a disjunction $c_1...c_k|w_{i+1}, ..., w_{j-1}$
22: **end for**

The next step is to repair the regex. Algorithm 1 takes as input a regex r, a word w, and a matching m. It runs left to right through the word (Line 1). For each mapped character position, it finds the next mapped character position (Line 2), and the lowest common ancestor (lca) of both leaf nodes in the syntax tree of r (Line 3). In a Kleene star, it may happen that the next character is mapped to a leaf node that lies to the left of the leaf node of the current character. Consider,

e.g., the regex `(abc)*` and the word "abfabc". Here, the second "a" is mapped to a leaf node that lies to the left of the node of "b": the links cross. In such a case, we find the next character that is mapped to the right of the current one (Line 5), and we add the obstructing substring as an optional concatenation (Line 7). In the example, we obtain `(ab(fab)?c)*`. If no such fix can be found, we remove the mapping (Line 10) and proceed. If the next character is mapped to a regex leaf node that succeeds the current leaf node, we trace the path from the left leaf node to the lca, and make all children to the right of that path optional (Lines 14-16). We proceed analogously with the right leaf node (Lines 17-19). Finally, we insert the unmatched substring below the lca (Lines 20-21).

3 Demo

Our demo allows the user to visualize the regex repairs that we propose. In the beginning, the user can enter a regex and a missing word. For example, the user can choose the regex `\d{2}/\d{2}/\d{4}`, and the missing word "1/8/1935". The demo then proceeds to show the user the maximal matching between the regex and the missing word, as computed by Myers' algorithm (Figure 1 (left)). In the following, the user can walk step by step through the modifications that our algorithm proposes. In the example, the algorithm will propose to make first the first digit optional, and then the third (Figure 1 right). Finally, the algorithm will simplify the regex to `(\d?\d/){2}\d{4}` (not shown in the figure).

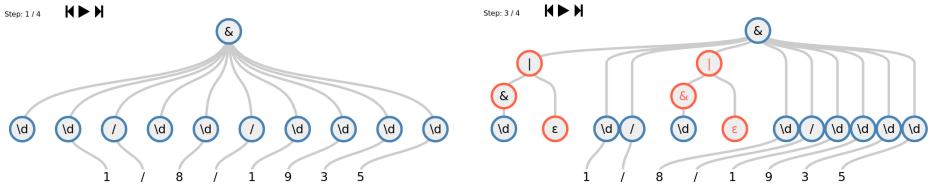

Fig. 1. Demo screenshots

Alternatively, the user can choose one of our pre-defined sets of words. We offer the ReLie dataset [4], a dataset of Wikipedia infoboxes (where 100 000 values of infobox attributes have been annotated by YAGO [9] with dates and numbers), and a sample of the Enron email dataset [5] (where we have annotated phone numbers manually). The user can then enter a regex that they think captures the words. We show the user which words are not captured, and the user can choose to add one of them to the regex. The user can then run the modified regex again to see how many more words are captured, and which ones are still missing. This way, the user can see how the algorithm performs on real data.

4 Experiments

Since information extraction for the Semantic Web is our intended use case, we ran our experiments on the afore-mentioned Wikipedia infobox dataset. We

asked 15 colleagues to manually produce regexes. We improve these regexes with our algorithm by randomly adding 10 words that the regexes did not match. We allowed the algorithm to reject the repair (and add the word in a disjunction instead) if the precision sinks on a training dataset. Table 1 compares the F1-value to the dis-baseline (which adds a word simply as a disjunction) and the star-baseline (`.*`). As expected, the dis-baseline raises recall only marginally. The star-baseline covers those infobox attributes that only consist of one match. Our approach, in contrast, can increase the recall of the regex. While these experiments cannot prove the viability of the approach under durance, they do show that automated regex repair is possible.

dataset	original			dis-baseline			star-baseline			repaired		
	R.	P.	F1	R.	P.	F1	R.	P.	F1	R.	P.	F1
YAGO/number	53.8	35.2	40.1	53.8	35.2	40.1	39.1	25.7	31.0	60.4	38.9	44.9
YAGO/date	68.9	81.3	70.1	68.9	81.3	70.1	48.4	26.6	34.3	72.0	81.5	72.2

Table 1. Recall (R.), Precision (P.), and F1 measure in percentage points.

5 Conclusion

We have presented an algorithm for automatically adding missing words to regular expressions. Our demo allows users to visualize and understand the changes we propose. For future work we plan to investigate other feedback functions, prevent overgeneralizing repairings and work on minimizing the length of the repaired regex. Our demo is available at `https://thomasrebele.org/projects/regex-repair`.

Acknowledgments. This research was supported by the grants ANR-11-LABEX-0045-DIGICOSME and ANR-16-CE23-0007-01 ("DICOS").

References

[1] Rohit Babbar and Nidhi Singh. "Clustering based approach to learning regular expressions over large alphabet for noisy unstructured text". In: *Workshop on Analytics for Noisy Unstructured Text Data*. 2010.

[2] Alberto Bartoli et al. "Automatic Synthesis of Regular Expressions from Examples". In: *IEEE Computer* 47.12 (2014).

[3] Falk Brauer et al. "Enabling information extraction by inference of regular expressions from sample entities". In: *CIKM*. 2011.

[4] Yunyao Li et al. "Regex learning for information extraction". In: *EMNLP*. 2008.

[5] Einat Minkov, Richard C Wang, and William W Cohen. "Extracting personal names from email". In: *EMNLP*. 2005.

[6] Karin Murthy, Deepak Padmanabhan, and Prasad Deshpande. "Improving Recall of Regular Expressions for Information Extraction". In: *WISE*. 2012.

[7] Eugene W Myers and Webb Miller. "Approximate matching of regular expressions". In: *Bulletin of mathematical biology* 51.1 (1989).

[8] Paul Prasse et al. "Learning to Identify Concise Regular Expressions That Describe Email Campaigns". In: *J. Mach. Learn. Res.* 16.1 (Jan. 2015).

[9] Fabian M. Suchanek, Gjergji Kasneci, and Gerhard Weikum. "Yago: a core of semantic knowledge". In: *WWW*. 2007.