Technical report: adding missing words to regular expressions

Rapport technique : ajout de mots manquants aux expressions régulières

Thomas Rebele
Katerina Tzompanaki
Fabian M. Suchanek

2018D002

Mars 2018

Département Informatique et Réseaux
LTCI : Groupe Data, Intelligence and Graphs

Technical Report: Adding Missing Words to Regular Expressions

Thomas Rebele[*], Katerina Tzompanaki[**], and Fabian M. Suchanek[*]

[*]*Telecom ParisTech, Paris, France*
[**]*Cergy-Pontoise University, Cergy, France*

March 28, 2018

Abstract

Regular expressions (regexes) are patterns that are used in many applications to extract words or tokens from text. However, even hand-crafted regexes may fail to match all the intended words.

In this paper, we propose a novel way to generalize a given regex so that it matches also a set of missing (previously non-matched) words. Our method finds an approximate match between the missing words and the regex, and adds disjunctions for the unmatched parts appropriately. We show that this method can not just improve the precision and recall of the regex, but also that it generates much shorter regexes than baselines and competitors on various datasets. This report complements our paper at the PAKDD 2018 conference. [18]

Rapport technique: Ajout de mots manquants aux expressions régulières

Résumé

Les expressions régulières (regex) sont des modèles utilisés dans de nombreuses applications pour extraire des mots ou des parties du texte. Cependant, même les regex faites à la main ne correspondent pas toujours à l'ensemble des mots prévus.

Dans cet article, nous proposons une nouvelle façon de généraliser une expression régulière donnée afin qu'elle corresponde également à un ensemble de mots manquants (précédemment non reconnus). Notre méthode trouve une correspondance approximative entre les mots manquants et l'expression regulière, et ajoute des disjonctions pour les parties non reconnues de façon appropriée. Nous montrons que cette méthode améliore la précision et le rappel de la regex, et aussi qu'elle génère des expressions regulières beaucoup plus courtes que l'approche naïve et que les algorithmes concurrents sur différents jeux de données. Ce rapport complète notre article soumis à la conférence PAKDD 2018. [18]

1 Introduction

Regular expressions (regexes) find applications in many fields: in information extraction, DNA structure descriptions, document classification, spell-checking, spam email identification, deep packet inspection, or in general for obtaining compact representations of string languages. To create regexes, several approaches learn them automatically from example words [2–5]. These approaches take as input a set of positive and negative example words, and output a regular expression. However, in many cases, the regexes are hand-crafted. For example, projects like DBpedia [10], and YAGO [20] all rely (also) on manually crafted regexes. These regexes have been developed by human experts over the years. They form a central part of a delicate ecosystem, and most likely contain domain knowledge that goes beyond the information contained in the training sets.

In some cases, a regex does not match a word that it is supposed to match. Take for example the following (simple) regex for phone numbers: `\d{10}`. After running the regex over a text, the user may find that she missed the phone number `01 43 54 65 21`. An easy way to repair the regex would be to add this number in a disjunction, as in `\d{10} | 01 43 54 65 21`. Obviously, this would be a too specific solution, and any new missing words would have to be added in the same way. A more flexible repair would *split* the repetition in the original regex and *inject* the alternatives, as in `(\d{2}\s?){4}\d{2}`.

In this paper, we propose an algorithm that achieves such generalizations automatically. More precisely, given a regex and a small set of missing words, we show how the regex can be modified so that it matches the missing words, while maintaining its assumed intention. This is a challenging endeavor, for several reasons. First, the new word has to be mapped onto the regex, and there are generally several ways to do this. Take, e.g., the regex `<h1>.*</h1>` and the word `<h1 id=a>ABC</h1>`. It is obvious to a human that the id has to go into the first tag. However, a standard mapping algorithm could just as well map the entire string ` id=a>ABC` onto the part `.*`. This would yield `<h1>?.*</h1>` as a repair – which is clearly not intended. Second, there is a huge search space of possible ways to repair the regex. In the example, `<h1(>.*| id=a>ABC)</h1>` is certainly a possible repair – but again clearly not the intended one. Finally, the repair itself is non-trivial. Take, e.g., the regex `(abc|def)*` and the word `abcabXefabc`. To repair this regex, one has to find out that the word is indeed a sequence of `abc` and `def`, except for two iterations. In the first iteration, the last character of `abc` is missing. In the second iteration, the first character of `def` has to be replaced by an `X`. Thus, the repair requires descending into the disjunction, removing part of the left disjunct and part of the right one, before inserting the `X` into one of them, yielding `(abc?|[dX]ef)*` as one possible repair.

Existing approaches typically require a large number of positive examples as input in order to repair or learn a regex from scratch. This means that the user has to come up with a large number of cases where the regex does not work as intended – a task that requires time, effort, and in some cases continuous user interaction (see Section 2 for examples). We want to relieve the user from this effort. Our approach requires not more that 10 non-matching words to produce meaningful generalizations. The contributions of this paper are as follows:

- we provide an algorithm to generalize a given regex, using string-to-regex matching techniques and adding unmatched substrings to the regex;
- we show how such repairs can be performed even with a small set of examples;
- we run extensive and comparative experiments on standard datasets, which show that our approach can improve the performance of the original regex in terms of recall and precision.

This paper is structured as follows. Section 2 starts with a survey of related work. Section 3 introduces preliminaries, and Section 4 presents our algorithm. Section 5 shows our experiments, before Section 6 concludes.

2 Related Work

In this paper we consider repairing regexes that fail to match a set of words provided by the user. We discuss work relevant to our problem along three axes: (1) matching regexes to strings, (2) automatic generation of regexes from examples, and (3) transformation of an existing regex based on examples.

Regex matching. Many algorithms (e.g., [6]) aim to match a regex efficiently on a text. Another class of algorithms deals with matching the input regex to a given input word – even though the regex does not match the string entirely [8,14]. Other algorithms for approximate regex matching [15,21] optimize for efficiency. We build on the seminal algorithm of [14]. Different from all these approaches, our work aims not just to match, but also to repair the regex.

Regex learning. Several approaches allow learning a regex automatically from examples. One approach [5] uses rules to infer regexes from positive examples for entity identifiers. Other work [2–4] uses genetic programming techniques to derive the best regex for identifying given substrings in a given set of strings. The work presented in [16] follows a learning approach to derive regexes for spam email campaign identification. In the slightly different context of combining various input strings to construct a new one, the work of [7] proposes a language to synthesize programs, given input-output examples. In the same spirit, the authors of [9] proposed an interactive framework in which users can highlight example subparts of text documents for data extraction purposes.

All of these works take as input a set of positive and negative examples, for which they construct a regex from scratch. In our setting, in contrast, we want to repair a given regex. Furthermore, we have only very few positive examples.

Regex transformation. There are several approaches that aim to improve a given regex. One line of work [11] takes as input a set of positive and negative examples as well as an initial regex to be improved. As in our setting, the goal is to maximize the F-measure of the regex. The proposed approach makes the regex stricter, so that it matches less words. Our goal is different: We aim to *relax* the initial regex, so that it covers words that it did not match before.

Similar to us, the work of [13] attempts to relax an initial regex. The approach requires the user in the loop, though, while our method is autonomous. Only two works [1,17] can relax a given regex automatically. However, as we will see in our experiments, both works produce very long regexes (usually over 100 characters). Our approach, in contrast, produces much shorter expressions – at comparable or even better precision.

3 Preliminaries

We assume that the reader is familiar with the basics of regexes. We write $L(r)$ for the language of a regex, and $T(r)$ for the syntax tree of a regex. Figure 1 shows an example of a regex with its syntax tree, along with a *matching* of a string to the regex. We further define a matching as follows:

Definition (Matching)**.** *Given a string s and a regex r, a* matching *is a partial function m from $\{1,...,|s|\}$ to leaf nodes of r's syntax tree $T(r)$, denoted $m|_{1,...,|s|}$, such that one of the following applies:*

- *r is a character or character class and $\exists i : s_i \in L(r)$*
- *$r = pq$ and $\exists i$ s.t. $m|_{1,...,i}$ is a matching for p and $m|_{i+1,...,|s|}$ is a matching for q*
- *$r = p\,|\,q$ and m is a matching for p or for q*
- *$r = p*$ and $\exists i_1, \ldots, i_j : i_1{=}1 \wedge i_1 < \cdots < i_j \wedge i_j{=}|s| + 1 \wedge \forall k \in \{1, \ldots, j\text{-}1\} : m|_{i_k,...,(i_{k+1})-1}$ is a matching for p*

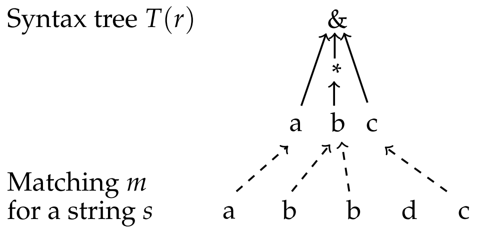

Figure 1: The syntax tree and a matching for the regex `ab*c`

A matching is *maximal*, if there is no other matching that is defined on more positions of the string. Figure 1 shows a maximal matching for the regex `ab*c` and the (non-matching) string `abbdc`. Maximal matchings can be computed with the algorithm proposed in [14]. The problem that we address is the following:

Problem Statement. *Given a regular expression r,a set S of positive examples, and a set E^- of negative examples, s.t. $|S| \ll |E^-|$, find a "good" regular expression r' s.t. $L(r) \subseteq L(r')$, $S \subseteq L(r')$, and $|L(r') \cap E^-|$ small.*

In other words, we want to generalize the regex so that it matches all strings it matched before, plus the new positive ones. For example, given a regex

$r =$ `[0-9]+` and a string $s =$"12 34 56", a possible regex to find is $r' =$`([0-9] ?)+`. This regex matches all strings that r matched, and it also matches s.

Now there are obviously trivial solutions to this problem. One of them is to propose $r =$`.*`. This solution matches s. However, it will most likely not capture the intention of the orginal regex, because it will match arbitrary strings. Therefore, one input to the problem is a set of negative examples E^-. The regex shall be generalized, but only so much that it does not match many words from E^-. The rationale for having a small set S and a large set E^- is that it is not easy to provide a large set of positive examples: these are the words that the hand-crafted regex does not (but should) match, and they are usually few. In contrast, it is somewhat easier to provide a set of negative examples. It suffices to provide a document that does not contain the target words. All strings in that document can make up E^-, as we show in our experiments.

Another trivial solution to our problem is $r' = r|s$. However, this solution will not capture the intention of the regex either. In the example, the regex $r' =$`[0-9]+|12 34 56` will match s, but it will not match any other sequence of numbers and spaces. Hence, the goal is to *generalize* the input regex appropriately, i.e., to find a "good" regex that neither over-specializes nor over-generalizes.

4 Repairing Regular Expressions

Given a regex r, a set of strings S, and a set of negative examples E^-, our goal is to extend r so that it matches the strings in S. Table 1 shows an example with only a single string $s \in S$ and no negative examples. Our algorithm is shown in Algorithm 1. It takes as input a regex r, a set of strings S, a set of negative examples E^-, and a threshold α. The threshold α indicates how many negative examples the repaired regex is allowed to match. Higher values for α allow a more aggressive repair, which matches more negative examples. $\alpha = 1$ makes the algorithm more conservative. In that case, the method will try to match at most as many negative examples as the original regex did. The algorithm proceeds in 4 steps, which we will now discuss in detail.

Original regex r:	`(\d{3}-){2}\d{4}`
String $s \in S$:	`(234) 235-5678`
Repaired regex:	`\(?\d{3}(-|\) )\d{3}-\d{4}`

Table 1: Example regex reparation

Step 1: Finding the Matchings. For each word $s \in S$, our algorithm finds the maximal matchings (see again Section 3). We use Myers' algorithm [14] for this purpose. The maximal matchings are collected in a set M.

Algorithm 1 Repair regex

```
INPUT: regex $r$, set of strings $S$, negative examples $E^-$, threshold $\alpha \geq 1$
OUTPUT: modified regex $r$
1: $M \leftarrow \bigcup_{s \in S} \mathit{findMatchings}(r, s)$
2: $\mathit{gaps} \leftarrow \bigcup_{m \in M} \mathit{findGaps}(m)$
3: $\mathit{findGapOverlaps}(r, \mathit{gaps})$
4: $\mathit{addMissingParts}(r, S, \mathit{gaps}, E^-, \alpha)$
```

Step 2: Finding the Gaps. The matching tells us which parts of the regex match the string. To fix the regex, we are interested in the parts that do *not* match the string. For this purpose, we introduce a data structure for the gaps in the string (i.e., for the parts of the string that are not mapped to the regex). Formally, a gap g for string s in a matching m is a tuple of the following:

- $g.start$: The index in the string where the gap starts (possibly 0).
- $g.end$: The index in the string where the gap ends (possibly `|s|+1`).
- $g.span$: The substring between $g.start$ and $g.end$ (excluding both).
- $g.m$: The matching m, which we store in the gap tuple for later access.
- $g.parts$: An (initially empty) set that stores sequences of concatenation child nodes. The sequences are disjoint, and partition the regex. Each $p \in g.parts$ is a possibility to inject $g.span$ into the regex as `(`p`|`$g.span$`)`.

> *Example (Finding the gaps):* When matching the word `(234) 235-5678` onto the regex `\d\d\d-\d\d\d-\d\d\d\d`, we encounter two gaps: gap_1 embraces the substring "`(`" with $gap_1.start = 0$, $gap_1.end = 2$. gap_2 embraces the substring "`) `" with $gap_2.start = 4$, $gap_2.end = 7$.

For each matching m, we find all gaps g that have no matching character in between (i.e., $\nexists k : g.start < k < g.end \wedge m(k)$ is defined), and where at least one of the following holds

- there is a character in the string between $g.start$ and $g.end$, i.e., $g.start < g.end - 1$
- there is a gap in the regex between $m(g.start)$ and $m(g.end)$, i.e., $m(g.start) \notin previous(m(g.end))$.

This set of gaps is returned by the method *findGaps* in Algorithm 1, Line 2.

Step 3: Finding Gap Overlaps. Gaps can overlap. Take for example the regex $r =$ `01234567` and the missing words `0x567` and `012y7`. One possible repair is `0(12)?(34|x|y)?(56)?7`. We can find this repair only if we consider the overlap between the gaps. In this example, we have two gaps: one with span `1234` and one with span `3456`. We have to partition the concatenation for the first gap into `12` and `34`, and for the second gap into `34` and `56`.

This is what Algorithm 2 does. It takes as input the regex r and the set of gaps *gaps*. It walks through the regex recursively, and treats each node of the regex. We split quantifiers $r\{min, max\}$ with $max < 100$ into $r\{\ldots\}r\{\ldots\}$, if the gap occurs between iterations. For other quantifiers, Kleene stars, and disjunctions, we descend recursively into the regex tree (Line 5).

For concatenation nodes, we determine all gaps that have their start point or their end point inside the concatenation (Line 7). Then, we determine the partitioning boundaries (Line 8; $l \in r$ means that regex r has a leaf node l). We consider each gap g (Line 9). We find whether the start point or the end point of any other gap falls inside g. This concerns only the boundaries between s and e (Lines 10-11). We partition the concatenation subsequence $c_s \dots c_{e-1}$ by cutting at the boundaries (Line 12). Finally, the method is called recursively on the children of the concatenation that contain the start point or the end point of any gap (Lines 13-14).

Algorithm 2 Find Gap Overlaps

```
INPUT: regex $r$, gaps $gaps$
OUTPUT: modified regex $r$, modified gaps $gaps$
1:  if $r$ is quantifier $q\{min, max\}$ with $max < 100$ then
2:      $r \leftarrow q\{\dots\}q\{\dots\}$ with appropriate ranges
3:      $findGapOverlaps(r, gaps)$
4:  else if $r$ is disjunction or Kleene star or quantifier then
5:      for child $c$ of $r$ do $findGapOverlaps(c, gaps)$
6:  else if $r$ is concatenation $c_1 \dots c_n$ then
7:      $gaps' \leftarrow \{g : g \in gaps \wedge g.m(g.start) \in r \vee g.m(g.end) \in r\}$
8:      $idx \leftarrow \{i+1 : g \in gaps' \wedge g.m(g.start) \in c_i\} \cup \{i : g \in gaps' \wedge$
    $g.m(g.end) \in c_i\}$
9:      for $g \in gaps'$ do
10:         $s \leftarrow$ i+1   if $g.m(g.start) \in c_i$, else 1
11:         $e \leftarrow$ i     if $g.m(g.end) \in c_i$, else $n+1$
12:         $g.parts \leftarrow g.parts \cup \{c_i \dots c_{j-1} : i, j \in idx \wedge s \leq i < j \leq e \wedge$
                                   $(\nexists k : k \in idx \wedge i < k < j)\}$
13:     for $c_i \in \{c_i : \exists g \in gaps'.\ g.m(g.start) \in c_i \vee g.m(g.end) \in c_i\}$ do
14:         $findGapOverlaps(c_i, gaps)$
```

Step 4: Adding Missing Parts. The previous step has given us, for each gap, a set of possible partitionings. In our example of the regex `01234567`, the word `012y7`, and the gap `3456`, we have obtained the partitioning `34|56`. This means that both `34` and `56` have to become optional in the regex, and that we can insert the substring `y` as an alternative to either of them: `012(34|y)(56)?7` or `012(34)?(56|y)7`. Algorithm 3 will take this decision based on which solution performs better on the set E^- of negative examples. It may also happen that none of these solutions is permissible, because they all match too many negative examples. In that case, the algorithm will just add the word as a disjunct to the original regex, as in `01234567|012y7`. To make these decisions, the algorithm will compare the number of negative examples matched by the repaired regex with the number of negative examples matched by the original regex. The ratio of these two should be bounded by the threshold α.

Algorithm 3 Add Missing Parts

```
INPUT: regex $r$, set of strings $S$, gaps $gaps$, negative examples $E^-$, threshold $\alpha \geq 1$
OUTPUT: modified regex $r$
1:  $org \leftarrow r$
2:  for g $\in$ gaps do
3:      for part p = $(c_i \cdots c_j) \in g.parts$ do
4:          $r' \leftarrow r$ with $c_i \cdots c_j$ replaced by $(c_i \cdots c_j | g.span)$,
                and all other parts $c_x \ldots c_y$ in $g.parts$ made optional with $(c_x \ldots c_y |)$
5:          if $|E^- \cap L(r')| \leq \alpha \cdot |E^- \cap L(org)|$ then
6:              $r \leftarrow r'$
7:              break
8:  $S' \leftarrow S \setminus L(r)$
9:  undo all changes for $s \in S'$ not required by other repairs
10: $G \leftarrow$ generalize words in $S'$
11: for g $\in$ G do
12:     if $|E^- \cap L(r|g)| \leq \alpha \cdot |E^- \cap L(org)|$ then
13:         $r \leftarrow r|g$
14:     else
15:         for $s \in L(g) \cap S'$ do
16:             $r \leftarrow r|s$
```

Algorithm 3 takes as input a regex r, a set of gaps $gaps$ with partitionings, negative examples E^-, and a threshold α. The algorithm first makes a copy of the original regex (Line 1) and treats each gap (Line 2). For each gap, it considers all parts (Line 3). In the example, we will consider the part `34` and the part `56` of the gap `3456`. The algorithm transforms the part into a disjunction of the part and the span of the gap. In the example, the part `34` is transformed into `(34|y)` (Lines 4). If the number of matched negative examples does not exceed the number of negative examples matched by the original regex times α (Line 5), the algorithm chooses this repair, and stops exploring the other parts of the gap (Lines 6-7). In Line 8, the algorithm collects all positive examples that are still not matched. The changes that were made for these words are undone (Line 9). Line 10 generalizes these words into one or several regexes. The generalization is adapted from [1]. First, we assign a group key to every word. The key is obtained by substituting substrings consisting only of digits with a (single) `\d`, lower or upper case characters with a `[a-z]` or `[A-Z]`, and remaining characters with character class `\W` or `\w`. Finally we obtain the group regex r by adding $\{min, max\}$ after every character class, such that r matches all strings in that group. The algorithm then checks if the regex obtained this way is good enough (Lines 12-13). If this is the case, the regex is added as a disjunction (Line 13). Otherwise the words that contributed to that group are added disjunctively

(Line 16). Table 1 shows how our method repairs the example regex.
Time complexity. We analyze the time needed to run our algorithm. Let M be the sum of the length of the missing words. Let N the length of the regex. Let N' the length of the expanded regex, where every quantifier $q\{min, max\}$ is replaced by $q \cdots qq? \cdots q?$. Furthermore let t be the runtime of applying a regex of length $O(N')$ on the negative examples E^-. Step 1 is a dynamic programming algorithm which fills up a table with M rows and $O(N')$ columns. It runs in $O(N'M)$ [14]. Step 2 iterates over the characters of the words. It checks two conditions, both in constant time reusing $previous(...)$ of step 1. Thus complexity of step 2 is $O(M)$. A word of length l can lead to at most $l + 1$ gaps. Therefore step 2 finds at most $O(M)$ gaps. Step 3 recursively descends into every node of the regex tree. For each of the concatenation nodes it might to iterate over $O(M)$ gaps. In doing so it produces at most $O(N)$ part boundaries per gap, therefore at most $O(N)$ parts per gap. The splitting of quantifier might lead to an exponential increase in the length of the regex. However, this happens also with the expansion of the regex in step 1, so the number of nodes that Algorithm 2 is limited by $O(N')$, so in total, step 3 runs in $O(N'M)$. Step 4 iterates over $O(M)$ gaps, each with at most $O(N)$ parts. Every part might require one application of the regex on the examples E^-. Therefore the first part of Algorithm 1 (Lines 1-8) runs in $O(NMt)$. Undoing (Lines 9-10) can be done in less than that time. The group keys for generalization can be obtained in $O(M)$, producing at most $O(M)$ generalized regexes. Applying those on the examples takes at most $O(Mt)$. In total, step 4 runs in time $O(NMt)$. In summary, our approach is dominated by adding the missing parts, and applying the candidates of repaired regexes on the examples. Its total runtime is thus $O(NMt + N'M)$.

5 Experiments

5.1 Setup

Measures. To evaluate our algorithm, we follow related work in the area [1,13, 17] and use a gold standard of positive example strings, $E^+ \supset S$. With this, the *precision* of a regex r is the fraction of positive examples matched among all examples matched:

$$prec(r) = \frac{|E^+ \cap L(r)|}{|L(r) \cap (E^+ \cup E^-)|}$$

The *recall* of r is the fraction of positive examples matched:

$$rec(r) = \frac{|E^+ \cap L(r)|}{|E^+|}$$

As usual, the *F1 measure* is the harmonic mean of these two measures.
Competitors. We compare our approach to both other methods [1,17] that can generalize given regexes (see Section 2). For [1], the code was not available upon request. We therefore had to re-implemented the approach. We think that

our implementation is fair, because it achieves a higher F1-value (87% and 84%, as opposed to 84% and 82%) when run on the same datasets as in [1] (s.b.) with a full E^+ as input.

Datasets. We use 3 datasets from related work. The *Relie dataset*[1] [11] includes four tasks (phone numbers, course numbers, software names, and URLs). Each task comes with a set of documents. Each document consists of a span of words, and 100 characters of context to the left and to the right. Each span is annotated as a positive or a negative example, thus making up our sets E^+ and E^-, respectively. We manually cleaned the dataset by fixing obvious annotation errors, e.g., where a word is marked as a positive and a negative example in the same task. In total, the dataset contains 90807 documents.

The *Enron-Random dataset*[2] [12] contains a set of emails of Enron staff. The work of [1] uses it to extract phone numbers and dates. Unfortunately, there is no gold standard available for these tasks, and the authors of [1] could not provide one. Therefore, we manually annotated phone numbers and dates on 200 randomly selected files, which gives us E^+. As in [1], any string that is matched on these 200 documents and that is not a positive example will be considered a negative example. In this way, we obtain a large number of negative examples.

The *YAGO-Dataset* consists of Wikipedia infobox attributes, where the dates and numbers that were used to build YAGO [20] have been annotated as positive examples. This dataset is used in [17]. As in *Enron-Random*, all strings that are not annotated as positive examples count as negative examples.

We thus have 8 tasks: 4 for the Relie dataset, 2 for the Enron dataset, and 2 for the YAGO dataset. Each task comes with positive examples E^+ and negative examples E^-. Our algorithm needs as input an initial regex that shall be repaired. For the Enron and YAGO tasks, we used the initial regexes given in [1, 17]. For Relie, we asked our colleagues to produce regexes by hand. For this purpose, we provided them with 10 randomly chosen examples from E^+ for each task, and asked them to write a regex. This gives us 5 initial regexes for each Relie task. Table 2 summarizes our datasets.

task	documents	avg. size	$\|E^+\|$	regexes
ReLie/phone [11]	41896	211	2657	5
ReLie/course [11]	569	210	314	5
ReLie/software [11]	44413	185	2307	5
ReLie/urls [11]	3929	176	735	5
Enron/phone [1]	225	1452	145	1
Enron/date [1]	225	1452	392	1
YAGO/dates [17]	100000	25	109824	15
YAGO/numbers [17]	100000	57	131149	15

Table 2: Statistics of the datasets

[1] `http://dbgroup.eecs.umich.edu/regexLearning/`

[2] `http://www.cs.cmu.edu/~einat/datasets.html`

Runs. Our algorithm does not take as input the entire set of positive examples E^+, but a small subset S of positive examples. To simulate a real setting for our algorithm, we randomly select S from E^+. We average our results over 10 different random draws of S. For each draw, we use each initial regex that we have at our disposal, and average our results over these. Thus, we run our algorithm 50 times for each Relie task, 10 times for each Enron-Random task, and 150 times for each YAGO task, and we average the obtained numbers over these. Our competitors are not designed to work on a small set of positive examples. Therefore, we provide them with additional positive examples obtained from running the original regex on the input dataset. Our method does not need this step. All algorithms are implemented in Java 8. The experiments were run on an Intel Xeon with 2.70GHz and 250GB memory.

		baseline		competitors		our approach		
task	original	dis	star	[1]	[17]	$\alpha = 1.0$	$\alpha = 1.1$	$\alpha = 1.20$
ReLie/phone	81.6	82.1	12.3	-.5 ▼	+.3	+.8	+1.6 △	**+2.3** ▲
ReLie/course	45.8	46.0	48.4	**+6.4** ▲	+.2 ▲	+.5 ▲	+1.3 ▲	+1.4 ▲
ReLie/software	9.2	12.4	9.9	+.1 ▲	-.0	+.7 ▲	+3.9	**+4.6**
ReLie/urls	55.2	56.0	31.5	-30.3 ▼	+.3	+2.9	**+4.2** ▲	**+4.2** ▲
Enron/phone	61.7	61.7	.1	-7.7	+5.3 △	**+21.0** ▲	**+21.0** ▲	**+21.0** ▲
Enron/date	72.3	72.4	.0	-49.6 ▼	-.0	**+.6**	**+.6**	+.4
YAGO/number	40.1	40.1	31.0	-11.5 ▼	+1.8 ▲	**+3.4**	+2.2	+2.4
YAGO/date	70.1	70.1	34.3	-26.3 ▼	+.3 ▲	**+6.9** ▲	+6.7 ▲	+6.6 ▲

Table 3: F1 measure for different values of the parameter α, improvement over the dis-baseline in percentage points. Bold numbers indicate the maximum F1 measure within each row. ▲ (and △) indicates significant improvement relative to the dis-baseline for a significance level of 0.01 (and 0.05).

		baseline		competitors		our approach		
task	original	dis	star	[1]	[17]	$\alpha = 1.0$	$\alpha = 1.1$	$\alpha = 1.20$
ReLie/phonenum	41.6	230.3	2.0	94.6	221.2	**46.8**	50.0	53.1
ReLie/coursenum	22.2	203.2	2.0	280.4	181.1	**33.7**	48.2	52.5
ReLie/softwarename	43.2	168.2	2.0	594.7	168.2	**54.4**	67.3	67.8
ReLie/urls	52.4	630.3	2.0	5826.1	570.2	**70.6**	74.7	74.7
Enron/phone	17.0	199.1	2.0	243.8	164.8	**41.4**	**41.4**	**41.4**
Enron/date	17.0	170.6	2.0	581.0	170.6	**34.2**	**34.2**	35.6
YAGO/number	65.4	223.2	2.0	19471.0	207.9	**119.4**	120.5	120.7
YAGO/date	191.4	336.2	2.0	4337.0	313.6	203.6	203.6	**203.5**

Table 4: Length of the repaired regexes (# of characters). Bold numbers indicate the shortest ones (without the original).

	competitors		our approach		
	[1]	[17]	$\alpha = 1.0$	$\alpha = 1.1$	$\alpha = 1.2$
dataset	total	total	total	total	total
ReLie/phonenum	.5	.3	2.3	2.3	2.4
ReLie/coursenum	.2	.2	.2	.1	.1
ReLie/softwarename	9.1	.6	3.1	2.8	2.8
ReLie/urls	1.6	4.4	1.9	2.0	1.6
Enron/phone	.4	1.5	1.1	1.0	1.0
Enron/date	.2	1.6	1.1	1.0	1.0
YAGO/number	16721.9	17.9	109.5	116.1	126.4
YAGO/date	396.5	5.8	58.3	54.1	58.0

Table 5: Runtime of the different algorithms (in seconds)

	$\alpha = 1.0$				$\alpha = 1.1$				$\alpha = 1.2$			
dataset	m.	f.	fb.	total	m.	f.	fb.	total	m.	f.	fb.	total
ReLie/phonenum	.2	.0	2.1	2.3	.1	.0	2.2	2.3	.1	.0	2.2	2.4
ReLie/coursenum	.1	.0	.1	.2	.1	.0	.0	.1	.1	.0	.0	.1
ReLie/softwarename	.3	.1	2.8	3.1	.2	.0	2.5	2.8	.3	.0	2.6	2.8
ReLie/urls	1.5	.1	.3	1.9	1.6	.1	.3	2.0	1.2	.1	.3	1.6
Enron/phone	.1	.0	1.0	1.1	.1	.0	.9	1.0	.1	.0	.9	1.0
Enron/date	.1	.0	1.0	1.1	.0	.0	.9	1.0	.0	.0	.9	1.0
YAGO/number	1.3	.0	108.1	109.5	1.4	.0	114.7	116.1	1.4	.0	125.0	126.4
YAGO/date	16.1	.0	42.2	58.3	15.7	.0	38.4	54.1	16.1	.0	41.8	58.0

Table 6: Runtime of the different phases of our algorithm (in seconds): matching (m.), fixing (f.), feedback (fb.), and for comparison the total time

5.2 Experimental Evaluation and Results

F1-measure. Table 3 shows the F1-measure on all datasets for different algorithms: the original regex, the disjunction-baseline (which consists just of a disjunction of the original regex with the 10 positive words), the star-baseline (which is just `.*`), the method from [1], the method from [17], and our method with different values for α. The table shows the improvement of the $F1$ measure w.r.t. the dis-baseline, in percentage points. For example, for *Relie/phone* and $\alpha = 1.2$, our algorithm achieves an F1 value of 81.6%+2.3%=83.9%. We can see that, across almost all tasks and settings, our algorithm outperforms the original regex as well as the dis-baseline. We verified the significance of the F1 measures with a micro sign test [22]. Detailed results on recall and precision can be found in our technical report [19].

If α is small, the algorithm is conservative, and tends towards the dis-baseline. If α is larger, the algorithm performs repairs even if this generates more negative examples. As we can see, the impact of α is marginal. We take this as an indication that our method is robust to the choice of the parameter.
Regex length. Table 4 shows the average length of the generated regexes (in number of characters). For our approach, the length depends on α: If the value is large, the algorithm will tend to integrate the words into the original regex. Then, the words are no longer subject to the generalization mechanism. Still, the impact of α is marginal: No matter the value, our algorithm produces regexes that are up to 8 times shorter than the dis-baseline, and nearly always at least twice as short as either competitor – at comparable or better precision and recall.
Runtime. For the ReLie and Enron dataset, all approaches take a time in the order of seconds for repairing one regex. Due to the much larger E^+, runtimes differ for the YAGO dataset: fastest system is [17] with 12 s on average, followed by our approach with 84 s. The runtime of our reimplementation of [1] lies in the order of minutes, as we did not optimize for runtime efficiency. Runtimes for all systems are shown in Table 5.

Table 6 shows the runtime for the different phases of the algorithm: the matching (Step 1), the fixing (Step 2-4), and the matching of the intermediate regexes on the set of negative examples E^- (which we call "feedback" in the table). In general, the runtime is in the order of seconds. For more complicated use cases (with longer regexes, longer missing words, and many negative examples), our method takes longer to execute. The matching phase takes more time if the regex contains nested quantifiers with a high maximal repetition number, as in `[a-z0-9]{0,30}){0,30}`. Most of the runtime is spent for the feedback. However, for reasonable inputs, our algorithm runs very fast: repairing a regex takes only some seconds.
Example. Table 7 shows an example of a repaired regex in the Relie/phonenum task. Our algorithm successfully identifies the non-matched characters `:` and `>` at the beginning of a phone number. It introduces them as options at the beginning of the original regex, leaving the rest of the regex intact. The dis-baseline, in contrast, would take the original regex, and add all words in a large disjunction. It is easy to see that our solution is more general and more

syntactically similar to the original regex.

Original regex:	`({0,1} \d{3}){0,1}(-|\.| ) \d{3}(-|\.| ) d{4}`			
Missing words:	`:734-763-2200`	`>317.569.8903`	`>443.436.0787`	`>512.289.1407`
`>734-615-9673`	`>734-647-8027`	`>734-763-5664`	`>734.647.3256`	`>773.339.3223`
Repaired regex:	`((|:|>)?\d{3})?(-|\.| )\d{3}(-|\.| )\d{4}`			

Table 7: Example of a scenario for the Relie/phonenum task

6 Conclusion

In this paper, we have proposed an algorithm that can add missing words to a given regular expression. With only a small set of positive examples, our method generalizes the input regex, while maintaining its structure. In this way, our approach improves the precision and recall of the original regex.

We have evaluated our method on various datasets, and we have shown that with few positive examples, we can improve the F1 measure on the ground truth. This is a remarkable result, because it shows that regexes can be generalized based on very small training data. What is more, our approach produces regexes that are significantly shorter than the baseline and competitors. This shows that our method generalizes the regexes in a meaningful and non-trivial way.

Both the source code of our approach and the experimental results are available online at `https://thomasrebele.org/projects/regex-repair`. For future work, we aim to shorten the produced regexes further, by generalizing the components into character classes. We hope that this line of research can make regexes ever more useful to their users.

References

[1] Babbar, R., Singh, N.: 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] Bartoli, A., Davanzo, G., Lorenzo, A.D., Mauri, M., Medvet, E., Sorio, E.: Automatic generation of regular expressions from examples with genetic programming. In: GECCO (2012)

[3] Bartoli, A., Davanzo, G., Lorenzo, A.D., Medvet, E., Sorio, E.: Automatic synthesis of regular expressions from examples. IEEE Computer 47(12) (2014)

[4] Bartoli, A., De Lorenzo, A., Medvet, E., Tarlao, F.: On the automatic construction of regular expressions from examples. In: GECCO (2016)

[5] Brauer, F., Rieger, R., Mocan, A., Barczynski, W.M.: Enabling information extraction by inference of regular expressions from sample entities. In: CIKM (2011)

[6] Ficara, D., Giordano, S., Procissi, G., Vitucci, F., Antichi, G., Di Pietro, A.: An improved dfa for fast regular expression matching. SIGCOMM Comput. Commun. Rev. 38(5), 29–40 (Sep 2008)

[7] Gulwani, S.: Automating string processing in spreadsheets using input-output examples. In: SIGPLAN Notices. vol. 46 (2011)

[8] Knight, J.R., Myers, E.W.: Approximate regular expression pattern matching with concave gap penalties. Algorithmica 14(1) (1995)

[9] Le, V., Gulwani, S.: FlashExtract: a framework for data extraction by examples. In: PLDI (2014)

[10] Lehmann, J., et. al: DBpedia - a large-scale, multilingual knowledge base extracted from wikipedia. Semantic Web J. 6(2) (2015)

[11] Li, Y., Krishnamurthy, R., Raghavan, S., Vaithyanathan, S., Jagadish, H.V.: Regular expression learning for information extraction. In: EMNLP (2008)

[12] Minkov, E., Wang, R.C., Cohen, W.W.: Extracting personal names from email: Applying named entity recognition to informal text. In: EMNLP (2005)

[13] Murthy, K., Padmanabhan, D., Deshpande, P.: Improving recall of regular expressions for information extraction. In: WISE (2012)

[14] Myers, E.W., Miller, W.: Approximate matching of regular expressions. Bulletin of mathematical biology 51(1) (1989)

[15] Navarro, G.: Approximate Regular Expression Searching with Arbitrary Integer Weights. Nord. J. Comput. 11(4) (2004)

[16] Prasse, P., Sawade, C., Landwehr, N., Scheffer, T.: Learning to identify concise regular expressions that describe email campaigns. J. Mach. Learn. Res. 16(1) (Jan 2015)

[17] Rebele, T., Tzompanaki, K., Suchanek, F.: Visualizing the addition of missing words to regular expressions. In: ISWC (2017)

[18] Rebele, T., Tzompanaki, K., Suchanek, F.: Adding missing words to regular expressions. In: PAKDD (2018)

[19] Rebele, T., Tzompanaki, K., Suchanek, F.: Technical report: Adding missing words to regular expressions. Tech. rep., Telecom ParisTech (February 2018)

[20] Suchanek, F.M., Kasneci, G., Weikum, G.: Yago: a core of semantic knowledge. In: WWW (2007)

[21] Wu, S., Manber, U., Myers, E.: A subquadratic algorithm for approximate regular expression matching. Journal of algorithms 19(3) (1995)

[22] Yang, Y., Liu, X.: A re-examination of text categorization methods. In: SIGIR (1999)

Dépôt légal : 2018 – 1er trimestre
Imprimé à Télécom ParisTech – Paris
ISSN 0751-1345 ENST D (Paris) (France 1983-9999)