Automatic String Replace by Examples

Andrea De Lorenzo
DIA - University of Trieste
Italy
andrea.delorenzo@phd.units.it

Eric Medvet
DIA - University of Trieste
Italy
emedvet@units.it

Alberto Bartoli
DIA - University of Trieste
Italy
bartoli.alberto@units.it

ABSTRACT

Search-and-replace is a text processing task which may be largely automated with regular expressions: the user must describe with a specific formal language the regions to be modified (search pattern) and the corresponding desired changes (replacement expression). Writing and tuning the required expressions requires high familiarity with the corresponding formalism and is typically a lengthy, error-prone process.

In this paper we propose a tool based on Genetic Programming (GP) for generating automatically both the search pattern and the replacement expression based only on examples. The user merely provides examples of the input text along with the desired output text and does not need any knowledge about the regular expression formalism nor about GP. We are not aware of any similar proposal. We experimentally evaluated our proposal on 4 different search-and-replace tasks operating on real-world datasets and found good results, which suggests that the approach may indeed be practically viable.

Categories and Subject Descriptors

I.5.4 [**Pattern Recognition**]: Applications—*text processing*; H.4.1 [**Information Systems Applications**]: Office Automation—*word processing*

Keywords

Search-and-replace, Genetic Programming

1. INTRODUCTION

Techniques for automated text processing are becoming increasingly important due to the uninterrupted growth and diffusion of text sources that are unstructured or loosely structured, e.g., logs—which exist in many different forms and application domains, including server administration, web access, phone calls, intrusion detection—web catalogs, email messages, social networking sites and so on. A specific text processing task potentially suitable to being automated is *search-and-replace*, where all text regions matching a given pattern should be replaced according to a given scheme. Many tools offer powerful support for this task based on the usage of *regular expressions*, which are a way of specifying a pattern using a formal language. The user must specify a *search pattern* for identifying the text regions to be modified and a separate *replacement expression* for describing the changes to be applied. The replacement expression usually include references to specific subregions of the region to be modified and the description of these subregions must be encoded in the search pattern appropriately. This framework requires the involvement of technically savvy users because defining the expressions required for solving a specific search-and-replace task requires high familiarity with the corresponding formalism. Furthermore, tuning the expressions is usually a time-consuming, error-prone process.

In this paper, we propose a tool based on Genetic Programming (GP) that is capable of *automatically* generating both the search pattern and the replacement expression, only by means of *examples*. The user merely provides a set of examples of the search-and-replace task, each example consisting of the text before and after the desired replacement without any further annotation. The tool then generates automatically both the search pattern and the replacement expression for fulfilling the task. The output can be used with popular processing engines, e.g., Java, PHP and so on. We emphasize that the user does not need to have any knowledge about GP nor is she required to provide any hints about the structure of the search pattern or replacement expressions to be obtained. We are not aware of any other method capable of automatically defining the required expressions, based solely on examples.

Our tool internally works in three phases. In the first phase, it executes a GP search for generating a regular expression able to localize the text regions to be processed. This regular expression defines a single pattern across all the provided examples, usually including the text to be modified and some surrounding text, thereby defining a sort of *context* for characterizing the scope of the desired replacements. For instance, consider the anonimization of Twitter usernames (e.g., `@GECCO2013` → `@xxxxxxxxx`): a letter should be replaced by `x`, but only when it is part of a Twitter username—which is the context. This phase is essential for making the user experience as simple as possible: the user merely specifies the input text t and the desired output text t'; she does not need to annotate t for indicating which of its portions have to be modified. The regular expression produced in the

GECCO'13, July 6–10, 2013, Amsterdam, The Netherlands.

first phase is required internally in the next phases and is not visible to the user. In the second phase, the tool builds the replacement expression by identifying the subregions to be modified and using references to those subregions appropriately, i.e., according to the provided examples. In the third and final phase, the tool executes a further GP search for generating the search pattern to be used along with the replacement expression generated at the previous step.

We evaluated our proposal on 4 search-and-replace tasks executed on different real-world datasets, each including several hundreds of manually-labelled examples. The tasks consist of anonymization of usernames in tweets, partial anonymization of IP addresses, format change of dates and phone numbers. The experimental evaluation shows that our tool is indeed able to define an effective search-and-replace task with only few tens of examples. Even leaving aside the potential of our tool for non technically savvy users, this result also suggests that evolutionary computing, coupled with the power of modern computing resources, may increasingly become a surrogate for some specific technical expertise of human specialists.

2. RELATED WORKS

Automatic generation of regular expressions from examples of the desired behavior may be useful in a variety of problems. We categorize these problems in increasing order of difficulty, as follows. (1) The *flagging* problem consists in assigning a binary label to a text: true, if some region of the text matches the regular expression, false otherwise. The examples consist of text regions, each accompanied by the respective desired label. (2) The *text extraction* problem consists in extracting from a text each region which matches the regular expression. The examples consist of texts, each accompanied by the annotation of the region to be extracted or the indication that nothing has to be extracted. (3) The *search-and-replace* problem is a generalization of the text extraction problem: the extracted regions have to be modified according to the specification encoded into an additional replacement expression. The examples consist of texts, each accompanied by the corresponding modified text or the indication that nothing has to be modified.

To the best of our knowledge, no method for automatic generation of expressions suitable for search-and-replace has been proposed before. Accordingly, we briefly review in this section: (i) approaches suitable for the flagging problem or the text extraction problem; (ii) proposals not based on regular expressions for facilitating users while performing search-and-replace tasks.

The authors of [5] propose an evolutionary approach for generating regular expressions for the flagging problem. Their approach is based on grammatical evolution and individuals are specified in the Backus-Naur Form. The method effectiveness is assessed on the task of identifying those lines of HTML documents which contain hyperlinks.

Several earlier works addressed the text extraction problem with evolutionary approaches, such as Genetic Algorithms (GA) [2, 9] and Genetic Programming (GP) [19, 7, 18, 3]. In [2], after an initial evolution, individuals are recombined and then selected to obtain the final regular expression. In [9], the alphabet for regular expressions is chosen after a preliminary frequency analysis on the set of examples. Both proposals are evaluated on the task of URL extraction (composition and size of training and testing sets are not provided in [9]). Methods based on GP encode regular expressions as GP trees and evolve the corresponding individuals—each individual being a candidate expression—to maximize effectiveness according to some metric [19, 7, 18].

The large computing power widely available today revamped GP-based solutions, allowing them to outperform earlier proposals and solve practically relevant problems [3]. In this work we solve a more general problem than in [3]—search-and-replace, rather than mere text extraction. Our proposal generates automatically, in the first and third phase, regular expressions for text extraction. The corresponding procedures are built according to the proposal in [3] but extend that proposal in several key aspects. First, part of the initial population contains individuals generated directly from the examples, rather than randomly. Second, we generate regular expressions including capturing groups, a feature which is needed for solving the search-and-replace problem (we describe this feature in the next section). Third, we use fitness definitions tailored to the specific requirements of search-and-replace. In the first phase, the evolutionary search promotes individuals describing a suitable context across all the examples, which is usually larger than the region to be extracted and modified. In the third phase, the fitness of an individual depends on the behavior of that individual when coupled with the replacement expression found in the second phase.

The scenario considered in [20] concerns criminal justice information systems and the goal consists in minimizing human effort for data mining. The proposed approach starts from a single example and produces a reduced form of regular expression exploiting the operator interventions during the learning process, which is hence not fully automatic. A similar scheme for regular expression generation which involves the human operator is presented in [10]: here an active learning algorithm is proposed which starts from a single example and then requires an external operator to respond to membership queries about candidate expressions. In our work, instead, we just require a set of examples and never require human involvement during the search.

Other promising proposals for the text extraction problem which are not based on evolutionary approaches have been presented in [11, 1, 4]. In [11] the user is required to provide a set of examples and an initial regular expression: the algorithm then applies successive transformations until it reaches a local optimum in terms of precision and recall. The system proposed in [1] works similarly but the authors focus on noisy data. The method proposed in [4] does not rely on an initial regular expression: instead, it identifies relevant patterns in the set of examples and then combines the most promising pattern into a single regular expression. The proposal is evaluated on several business-related text extraction tasks, e.g., phone numbers and invoice numbers.

Concerning proposals for facilitating users in automated text editing that are not based on regular expressions, the authors of [14] propose a system (LAPIS) based on a pattern language. This pattern language, previously proposed by the same authors [16], can replace regular expressions in many common tasks, including simple forms of search-and-replace. Since some skill is still required to use the language, LAPIS offers an assisted mode in which an initial pattern is inferred from a set of positive and negative examples. Differently from our work, the assisted mode addresses only the search

portion of the search-and-replace task. Furthermore, the results produced by our tool are not bound to a specific text processing system but can be used in a wealth of different environments. A similar scheme for inferring a pattern from examples is used in [15]: the goal here is to guess multiple selections for simultaneous editing.

A method for assisting the user in executing a search-and-replace tasks is proposed in [13]. The authors consider a scenario where, in order to mitigate the difficulty of defining a search pattern—which is the point we address in this paper—users often work with less precise patterns and then manual check each suggested match. They propose to cluster the suggested matches so as to reduce the number of manual checks, since the user approve or reject the whole cluster instead of inspecting each single match.

3. OUR APPROACH

The user provides a set of examples T. Each example is composed by a pair of strings $\langle t, t' \rangle$, where t is a string to be modified in t'. An example in which $t' = t$ is called a *negative* example.

The output of our system is a pair of strings $\langle s, r \rangle$, where s is the regular expression which defines the search pattern and r is the replacement expression.

The regular expression s may contain zero or more *capturing groups*. A capturing group is a substring of s enclosed between round parentheses. A capturing group is itself a regular expression: when a string t matches a regular expression s containing a capturing group, a substring of t matches the capturing group. The substring matched by a capturing group can be referenced in the replacement expression. The corresponding syntax is `$`n[1], where n is the index of occurrence of the capturing group in the regular expression—e.g., `$1` indicates the substring matched by the first capturing group. For example, suppose the user needs to change the date format from month-date-year to day-month-year: a suitable search pattern, which includes three capturing groups, is the regular expression `(\d+)-(\d+)-(\d+)` and the corresponding replacement expression is `$2-$1-$3`.

Solving the search-and-replace problem by means of regular expressions requires a notion of *context*. In general, a single pattern identifying only the substrings to be replaced might not exist. That is, it might not exist a single regular expression which, for each example $\langle t, t' \rangle$, exactly matches the shortest substring of t including the characters which have to be modified in order to obtain t'. In practice, however, a pattern may often be found for superstrings of the strings to be replaced: we call these superstrings the *context* and this pattern the *context pattern*.

To clarify, the first two columns of Table 1 show a few examples related to three different search-and-replace tasks (the other columns show intermediate results discussed later). The third column illustrates the substring of t that is to be replaced: it is apparent that, for each of the three tasks, there is not any pattern identifying these substrings. On the other hand, a context pattern as defined above does exist for each of the three tasks: column c_k contains the contexts—i.e., superstrings of the substring to be replaced—extracted by the context pattern in column s_c.

[1]Some regular expression engines (e.g., Python, .NET, Ruby) use the $\backslash n$ notation, instead of the `$`n notation used, e.g., in Java, JavaScript, PHP, ...)

The context thus describes portions of input that do have to be replaced and where replacements have to be confined. An essential component of our approach is that we do *not* require the user to specify the context. On the contrary, the context is discovered automatically by the system. For instance, in the third row in Table 1, we do not require that the user specifies that `ic` has to be replaced by `xx` only when `ic` is part of the Twitter username `@Toxic`, but not when is part of `Sick`. It is up to the system to extract that information from the examples. The user only describes the input text and the desired output.

Note that, for a given set of examples, the context pattern may not be determined unambiguously—e.g., in Table 1, each context could also start with the space character, or with multiple arbitrary characters followed by a space. A given set of examples might thus be associated with 0, 1 or more context patterns and finding those patterns is not straightforward.

Our proposal consists of three phases, described in the next sections in full detail: (i) generate a context pattern s_c, (ii) build the replacement expression r, (iii) generate the search pattern s which works with r. We remark that phase 1 is necessary because we decided to not rely on the user for specifying the context of the desired changes. The result s_c of this phase is not exposed to the user and is only an input for the next phase.

4. IMPLEMENTATION

4.1 Generating the context pattern

In this phase we aim at generating the context pattern s_c. To this end, we build from T a learning set T_c suitable for a text extraction problem: for each example $\langle t, t' \rangle$ in T we construct exactly one example $\langle t, D(t, t') \rangle$ for T_c, where $D(t, t')$ is the substring of t to be replaced and is determined as described below. Then, we run a GP search for generating a regular expression that attempts to satisfy the examples in T_c.

The string $D(t, t')$ is the shortest substring of t which includes all characters that have to be modified in order to obtain t' (see also the third column in Table 1). Formally, let t^i be the i-th character in t and let $\mathcal{L}(t)$ be the length of t. Then $D(t, t') = t^i t^{i+1} \dots t^{\mathcal{L}(t)-j}$, where $i \geq 1$ is the lowest integer for which $t^i \neq t'^i$ and $j \geq 0$ is the lowest integer for which $t^{\mathcal{L}(t)-j} \neq t'^{\mathcal{L}(t')-j}$; if $t = t'$, $D(t, t') = \emptyset$.

For the purpose of the GP search, we partition the learning set T_c in a training set T_c^t and a validation set T_c^v. We form these two sets so as to distribute the negative examples evenly.

We then run a GP search on T_c^t, as follows. Every individual is a tree which represents a regular expression. The *function set* consists of (we assume the reader has some familiarity with regular expressions [8]): (i) the *concatenator*, that is a binary node that concatenates other nodes or leaves, (ii) the *character class* operators `[·]` and `[^·]`, (iii) the *capturing group* `(·)` and the *non-capturing group*[2] `(?:·)` operators, (iv) the *possessive quantifiers* (`·*+`, `·++`, `·?+`, `{·,·}+`). The *terminal set* consists of: (i) *constants*—i.e., a single character, a number or a string, (ii) *ranges*—i.e., `a-z` or `A-Z` and (iii) *character classes*, i.e., `\w` or `\d`.

[2]A non-capturing group is a group which cannot be referenced in the replacement expression.

Table 1: Three sets of synthetic examples and corresponding intermediate and final results: $\emptyset$ indicates the empty string.

t_k	t'_k	$D(t_k, t'_k)$	s_c	c_k	c'_k	r_k	s	r
I like @GECCO13 conf	I like @GExxxx conf	CCO13		@GECCO13	@GExxxx	$1xxxx		
RT @MaleLabTs New paper	RT @Maxxxx New paper	leLabTs	@\w\w(\w+)	@MaleLabTs	@Maxxxx	$1xxxx	(@\w\w)\w+	$1xxxx
Sick of @Toxic chatter	Sick of @Toxxxx chatter	ic		@Toxic	@Toxxxx	$1xxx		
nothing new here	nothing new here	$\emptyset$		$\emptyset$	$\emptyset$	$\emptyset$		
today is 1-23-13	today is 23/01/13	1-23-		1-23-13	23/1/13	$2/$1/$3		
he left on 3-13-12	he left on 13/3/12	3-13-	\d+-\d+-\d+	3-13-12	13/3/12	$2/$1/$3	(\d+)-(\d+)-(\d+)	$2/$1/$3
great 1-1-13 party!	great 1/1/13 party!	-1-		1-1-13	1/1/13	$1/$2/$3		
he is Nick	he is *Nick*	Nick	[A-Z]\w+	Nick	*Nick*	*$1*	([A-Z]\w+)	*$1*
John was here	*John* was here	John		John	*John*	*$1*		

We build the initial population of P individuals so that half of them are generated directly from the examples, rather than randomly. If the number of examples is smaller than $\frac{P}{2}$, then we use them all and generate the remaining individuals randomly. In detail, let $P_E = \min\left(\frac{P}{2}, 2|T_c^t|\right)$, where $|T_c^t|$ is the size of T_c^t, and let $\langle t_k, D(t_k, t'_k)\rangle$ denote the k-th example in T_c^t. Then, the initial population consists of: $\frac{P_E}{2}$ individuals, each representing one of the strings $D(t_k, t'_k)$; $\frac{P_E}{2}$ individuals, each representing one of the strings $D(t_k, t'_k)$ where each digit is replaced with a `\d` and each alphabetical character with a `\w`; the remaining $P - P_E$ individuals are generated at random with a Ramped half-and-half method. We chose this strategy because we found, after preliminary experimentation, that it greatly speeds up the convergence toward good solutions.

The *fitness* function to be minimized during the search is the sum of the Levenshtein distances[3], across all the examples in the training set, between the string to be extracted and the string actually extracted by the first capturing group:

$$f_E(s_c) = \sum_{k=1}^{|T_c^t|} L(D(t_k, t'_k), E_1(t_k, s_c)) \qquad (1)$$

where s_c is the evaluated individual, $L(x, y)$ is the Levenshtein distance between strings x and y, $E_1(t_k, s_c)$ is the substring of t_k matched by the first capturing group of s_c—if s_c does not contain a capturing group, we set $f_E(s_c) = +\infty$. We designed the fitness based on the key requirement of this phase, that is, automatic generation of a single pattern capable of extracting a superstring of the substring to be modified. For this reason, we promote individuals with at least one capturing group and such that this group matches $D(t_k, t'_k)$. We did not include in the fitness any component depending on what is extracted beyond $D(t_k, t'_k)$ because, as discussed in the previous section, we have no explicit information about what the whole expression should extract, i.e., about the context.

We run N_1 independent GP searches, differing only for the random seed. We evaluate on the validation set T_c^v each of the N_1 resulting regular expressions, and select as output s_c of this phase the one which minimizes f_E.

4.2 Building the replacement expression

[3]The Levenshtein distance measures the difference between two strings: informally, it corresponds to the minimum number of single-character edits required to change one string into the other.

In this phase we aim at generating the replacement expression r. To this end, we generate a candidate replacement expression r_k for each positive example $\langle t_k, t'_k\rangle$ in T, as follows.

Let c_k be the substring of t_k extracted by the context pattern s_c; let b, e be the integers such that $c_k = t_k^b t_k^{b+1} \dots t_k^{\mathcal{L}(t_k)-e}$; let c'_k be the substring of t'_k delimited in the same way by b and e, i.e., $c'_k = t'^b_k t'^{b+1}_k \dots t'^{\mathcal{L}(t'_k)-e}_k$ (if $b \geq \mathcal{L}(t'_k) - e$, then $c'_k = \emptyset$). We set the candidate replacement expression r_k for the example $\langle t_k, t'_k\rangle$ by executing Algorithm 1, which takes as input c'_k and the list of tuples C_k constructed with the following steps. Intuitively, C_k describes the boundaries of all the (maximal) substrings of c_k which appear in c'_k. In detail, we (i) construct the list C_k containing all tuples $\langle i, j, i', j'\rangle$ such that $c_k^i \dots c_k^j = c'^{i'}_k \dots c'^{j'}_k$; (ii) remove from C_k each tuple $\langle i, j, i', j'\rangle$ for which there exists another tuple $\langle i^*, j^*, i'^*, j'^*\rangle$ such that $i \geq i^*$ and $j \leq j^*$; (iii) sort C_k according to index i, in ascending order, and insert into each tuple an increasing integer n that represents the position of the tuple in C_k; (iv) sort C_k according to index i', in ascending order. At this point, we set $r_k = \mathcal{R}(c'_k, C_k)$ where $\mathcal{R}$ is defined in Algorithm 1. As pointed out above, C_k describes the boundaries of the substrings of c_k which appear in c'_k—in brief, of the common substrings. Algorithm 1 builds r_k by concatenating the substrings of c'_k between two common substrings, replacing common substrings by tokens $\$n$, after an appropriate sorting—see Figure 1 for an example.

Algorithm 1 Algorithm for building a candidate replacement expression r_k.

function $\mathcal{R}(c', C = \{\langle i_1, j_1, i'_1, j'_1, n_1\rangle, \dots\})$
  $r \leftarrow \emptyset$
  $b \leftarrow 1$
  $e \leftarrow i'_1 - 1$
  for $h \leftarrow 1, |C| - 1$ **do**
    $r \leftarrow r c'^b \dots c'^e \$n_h$ ▷ concatenation
    $b \leftarrow j'_h + 1$
    $e \leftarrow i'_{h+1} - 1$
  end for
  $r \leftarrow r c'^b \dots c'^e \$n_{|C|} c'^{j'_{|C|}+1} \dots c'^{\mathcal{L}(c')}$ ▷ concatenation
  return r
end function

Different examples might generate different replacement expressions, owing to conflicting or ambiguous examples. For

Figure 1: Example of execution of $\mathcal{R}(c'_k, C_k)$ where C_k has been constructed from c_k, c'_k

c_k = `07-14-1789`

c'_k = `<b>14/07/1789</b>`

C_k=

i,	j,	i',	j',	n	
$\langle 4,$	$5,$	$4,$	$5,$	$2\rangle$	`14` → `$2`
$\langle 1,$	$2,$	$7,$	$8,$	$1\rangle$	`07` → `$1`
$\langle 7,$	$10,$	$10,$	$13,$	$3\rangle$	`1789` → `$3`

r_k = `<b>$2/$1/$3</b>`

instance, in Table 1, the third example generates a replacement expression `$1xxx` different from the one corresponding to the first and second example `$1xxxx`. The reason is the ambiguity associated with the `x` character, which is both part of the input text to be modified (`Toxic`) and of the text to be obtained (`Toxxxx`).

We select as output of this phase the replacement expression r which occurs most frequently among all the examples. In case two or more candidates occur the same number of times, we choose one of them at random.

4.3 Generating the search pattern

In this phase, we aim at generating the search pattern s. To this end, we partition the set of examples T in a training set T^t and a validation set T^v, by distributing negative examples evenly, and execute a GP search similarly to Section 4.1 with a crucial difference in the fitness definition. In this case we associate with each individual, i.e., regular expression, s two objective functions to be minimized:

$$f_R(s) = \sum_{k=1}^{|T^t|} L(t'_k, R(t_k, s, r)) \quad (2)$$

$$f_G(s) = |G_s(s) - G_r(r)| \quad (3)$$

where $R(t_k, s, r)$ is the string obtained by performing on t_k the search-and-replace task defined by the regular expression s and the replacement expression r (found in the previous phase), $G_s(s)$ is the number of capturing groups defined in s and $G_r(r)$ is the number of capturing groups defined in r. We minimize this multi-objective fitness by means of NSGA-II [6].

We run N_2 independent GP searches, differing only for the random seed, thereby obtaining N_2 search patterns s_h. Finally, we evaluate on the validation set T^v each of the N_2 candidate solutions $\langle s_h, r\rangle$ for the search-and-replace task, and select as final result the one which minimizes f_R and f_S by means of NSGA-II.

5. EXPERIMENTS

We experimentally evaluated our proposal on real-world datasets that we manually annotated for 4 search-and-replace tasks:

Twitter anonymization Replace each username found in a tweet corpus with `@xxxxxx`—e.g., `@GECCO2013` becomes `@xxxxxx`. The tweet corpus has been taken from [12];

Table 2: GP parameters

Parameter	Settings
Population size (phase 1)	500
Population size (phase 3)	3000
Number of generations (phase 1)	1000
Number of generations (phase 3)	200
Selection	Tournament of size 7
Initialization depths	1–5
Max depth after crossover	15
Reproduction rate	10%
Crossover rate	80%
Mutation rate	10%

IP partial anonymization Replace the second two digit groups of each IP address (expressed in dot-decimal notation) found in a web server log with `xxx.xxx`—e.g., `127.0.0.1` becomes `127.0.xxx.xxx`.

Date format change Change each date found in the web server log of the previous task from the Gregorian little-endian slash separated format to the Gregorian big-endian dash separated format—e.g., `31/Dec/2012` becomes `2012-Dec-31`.

Phone number format change Change each phone number found in an email collection by removing the parenthesis around the area code and adding a dash—i.e, `(555) 555-5555` becomes `555-555-5555`. The email corpus has been taken from [17] and was used by [11, 4].

The dataset consists of 1000 examples for each task, i.e., 1000 pairs $\langle t, t'\rangle$, of which 500 are negative. We executed three experiments for each task, varying the size of the set of examples.

We executed each experiment as follows: (i) we randomly split the dataset in two subsets T and T^e; each subset is balanced, i.e., contains the same number of positive and negative examples; (ii) we generated a solution $\langle s, r\rangle$ using T and evaluated the solution on T^e—i.e., T is the learning set and T^e is the testing set (during phases 1 and 3 the learning set is further split in training and validation, as discussed in the previous sections). We report results obtained with 5-fold cross-validation, i.e., we repeated the two steps above 5 times and averaged the performance indexes exhibited by the 5 solutions on the testing set of the corresponding experiment. We set the parameters for the GP searches as in Table 2. We chose the number of generations and the population size, which are different between phase 1 and phase 3, after some preliminary experimentation. We chose an equal number of independent searches in phase 1 and phase 3: $N_1 = N_2 = 32$.

We evaluated each generated solution $\langle s, r\rangle$ by means of two metrics. The *distance error rate* ϵ_d quantifies the percentage of characters that have not been processed correctly, i.e., it averages on T^e pairs the distance between the expected string and the string actually obtained, divided by the length of the former. The *count error rate* ϵ_c quantifies the percentage

Table 4: Experiment execution times, averaged across the 5 repetitions: third to fifth columns show the average execution of the three phases.

Task	$\lvert T^t\rvert + \lvert T^v\rvert$	Time (min)			
		1	2	3	Overall
Twitter anonymization	20	0	0	1	1
	25	0	0	1	1
	50	0	0	2	2
IP partial anonymization	20	2	0	10	12
	25	7	0	26	33
	50	4	0	39	43
Date format change	20	3	0	15	18
	25	7	0	29	36
	50	13	0	65	78
Phone number format change	20	7	0	28	35
	25	17	0	63	80
	50	27	0	105	132

of T^e pairs that have not been processed correctly. In detail:

$$\epsilon_d = \frac{1}{|T^e|} \sum_{i=1}^{|T^e|} \frac{L(t'_k, R(t_k, s, r))}{\mathcal{L}(t'_i)} \quad (4)$$

$$\epsilon_c = \frac{1}{|T^e|} \sum_{i=1}^{|T^e|} \mathbb{1}(t'_k, R(t_k, s, r)) \quad (5)$$

where $\mathbb{1}(x, y)$ is a function whose value is 1 if x and y are equal, 0 otherwise.

5.1 Results

The salient results are summarized in Table 3. Each row corresponds to one experiment and reports the results in terms of ϵ_d and ϵ_c, with average μ and standard deviation σ across the 5 repetitions. For ϵ_d, we show also the values for each repetition. We remark that the learning set $T = T^t \cup T^v$ is always a small portion of the dataset, less than 10%.

It seems fair to claim that the approach does provide very good performance. A set of 50 examples suffices to execute the "IP partial anonymization" and "Date format change" tasks without any mistake, which seems to be a remarkable result. Furthermore, 50 examples for the "Twitter anonymization" task suffice to achieve correct processing of 96.9% of the testing set instances. Concerning the "Phone number format change" task, the percentage of testing instances processed correctly is 92%, again as long as 50 or more examples are available for the learning procedure. To place this result in perspective we observe that the dataset for this task has been used in earlier works addressing automatic generation of solutions for the text extraction problem from examples [11, 4]. The cited works used training sets much bigger than ours. The results were provided in terms of F-measure and range in 85%–87% with 4100–33400 learning examples for [11] and 65%–92% with 400–52000 learning examples for [4]. Although our indexes cannot be compared directly to F-measure, which is not meaningful in our context, our ability of processing more than 92% of the testing instances correctly, even with a learning set smaller by one order of magnitude or more, seems to be a good result.

The previous results are the average performance across 5 repetitions of each experiment, where the result of each experiment is the best solution (on the learning set) across the 32 independent GP searches in phase 3. Further insights can be obtained by analyzing the performance of all the $5 \times 32 = 160$ solutions found for each task. To this end, Figure 3 plots the number of solutions with $\epsilon_d < 10\%$ and $\epsilon_c < 10\%$ (on the testing set). Each bar corresponds to an experiment repetition. It can be seen that our approach does generate a number of good solutions systematically, that is, the good performance is not the result of a single lucky individual.

It is also interesting to point out that there is a clear correlation between performance of an individual on the validation set T^v and its performance on the testing set T^e: Figure 2 shows ϵ_d on the two sets, for each of the $5 \times 32 = 160$ solutions found in our experiments (one plot per task). This outcome demonstrates that the relative performance on individuals on the validation set is a good predictor of their relative performance on the testing set.

Table 4 reports the execution time for each experiment, averaged across the 5 repetitions, with the indication of the time taken by each of the three phases. Each experiment has been executed in parallel on 4 identical machines powered with a quad-core Intel Xeon X3323 (2.53 GHz) and 2GB of RAM. The execution times are, in most cases, too high to devise a possible interactive use of our approach. On the other hand, they seem to be sufficiently low to be practical, especially in a not far away future. Besides, the corresponding computing effort might be leased at 1 or 2 USD per hour[4]—a less accurate but much cheaper surrogate for the specific skills of a specialist.

6. CONCLUDING REMARKS

We have considered the feasibility of solving a search-and-replace task described solely through examples by means of regular expressions. The motivation for this problem follows from the ever increasing wealth of unstructured or loosely structured text sources, along with the need of automated techniques for their processing.

We have presented the implementation of a tool able to generate the required search pattern and replacement expression automatically. The user merely provides examples of the input text coupled with the desired output text, without any further annotation or hints about the expected results. We are not aware of any other proposal with these features.

We assessed the performance of our tool on challenging search-and-replace tasks executed on real-world datasets. The experimental evaluation provided very good results and suggests that the approach may indeed be practically viable.

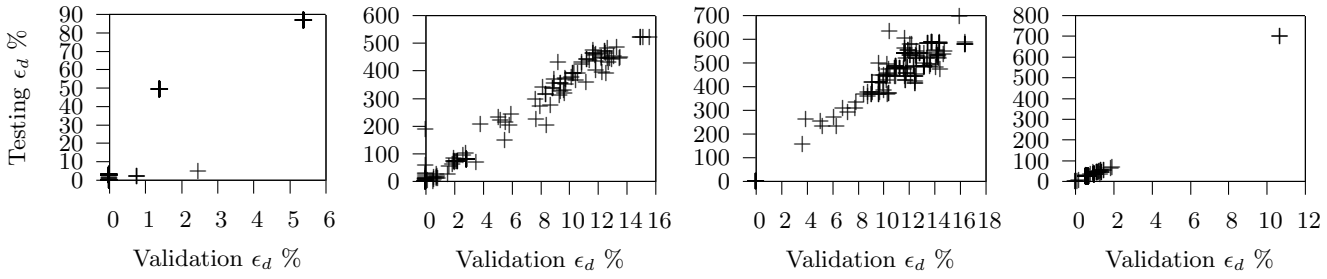
7. REFERENCES

[1] R. Babbar and N. Singh. Clustering based approach to learning regular expressions over large alphabet for noisy unstructured text. In *Proceedings of the fourth workshop on Analytics for noisy unstructured text data*, AND '10, pages 43–50, New York, NY, USA, 2010. ACM.

[2] D. Barrero, D. Camacho, and M. R-Moreno. Automatic Web Data Extraction Based on Genetic Algorithms and Regular Expressions. *Data Mining and Multi-agent Integration*, pages 143–154, 2009.

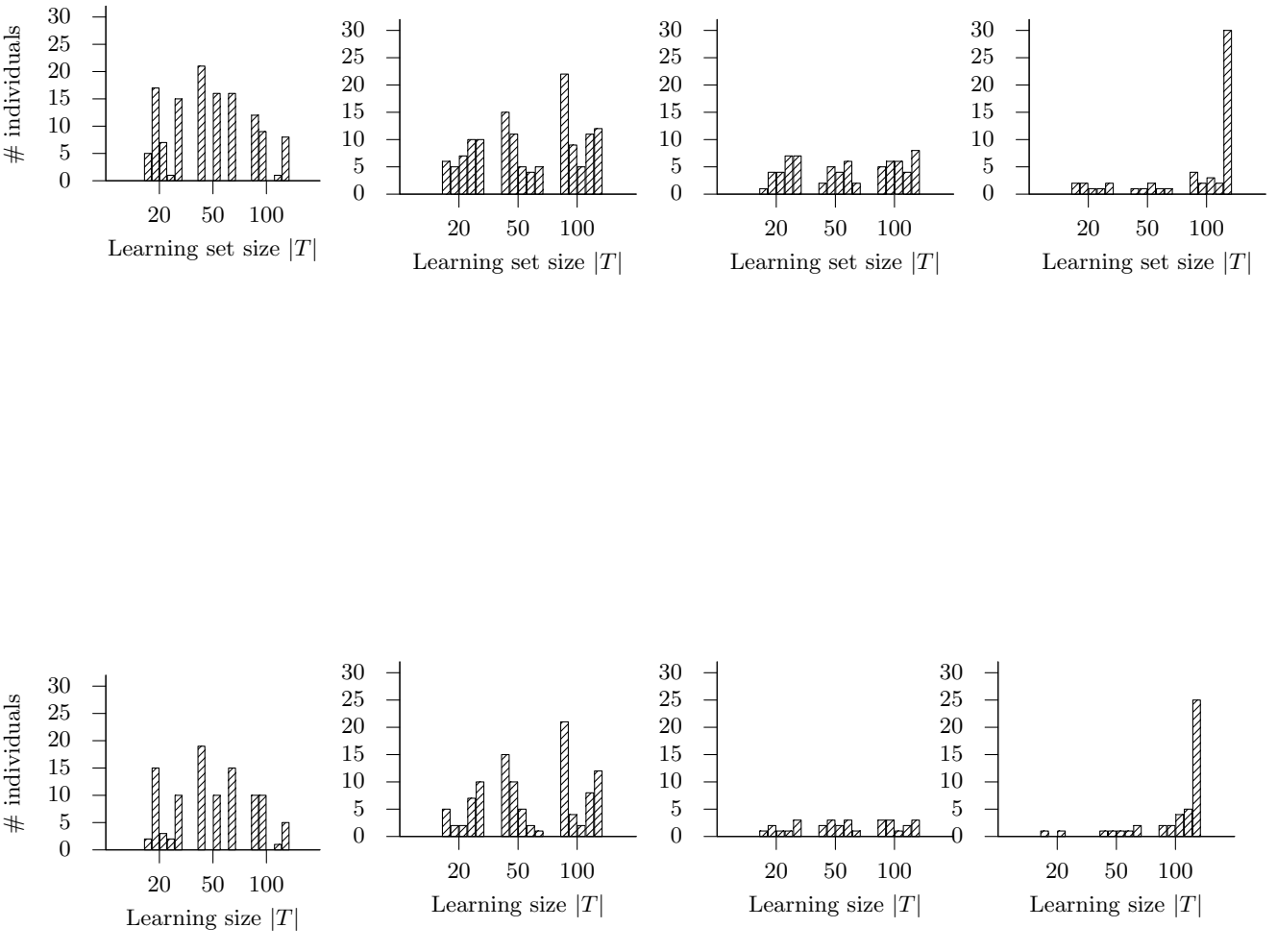
[3] A. Bartoli, G. Davanzo, A. De Lorenzo, M. Mauri, E. Medvet, and E. Sorio. Automatic generation of

[4] `http://aws.amazon.com/ec2/pricing/`



(a) Twitter anonymization (b) IP partial anonymization (c) Date format change (d) Phone n. format change

Figure 2: Validation ϵ_d vs. testing ϵ_d.



(a) Twitter anonymization (b) IP partial anonymization (c) Date format change (d) Phone n. format change

Figure 3: Number of generated solutions with $\epsilon_d \leq 10\%$ (above) and $\epsilon_c \leq 10\%$ (below). Each bar corresponds to a repetition.

Table 3: Experiment results

Task	Dataset			Repetition (ϵ_d %)					Overall (ϵ_d %)		Overall (ϵ_c %)	
	$\|T^t\|$	$\|T^v\|$	$\|T^e\|$	1	2	3	4	5	μ	σ	μ	σ
Twitter anonymization	10	10	980	1.7	0.0	7.7	10.7	0.0	4.0	4.9	5.5	10.5
	25	25	950	0.0	53.9	0.0	14.0	0.0	13.6	23.3	3.1	3.7
	50	50	900	0.0	0.0	14.0	9.5	0.0	4.7	6.6	2.0	1.6
IP partial anonymization	10	10	980	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.5	0.7
	25	25	950	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	50	50	900	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Date format change	10	10	980	32.7	58.5	58.1	0.0	0.0	29.9	29.2	60.0	54.8
	25	25	950	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	50	50	980	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Phone number format change	10	10	980	7.2	4.7	6.7	6.2	7.1	6.4	1.0	52.4	4.13
	25	25	950	1.9	0.9	0.0	0.0	12.9	3.2	5.5	8.2	10.8
	50	50	900	12.3	0.0	0.0	2.8	0.0	3.0	5.3	6.6	11.2

regular expressions from examples with genetic programming. In *Proceedings of the 14th GECCO conference companion*, pages 1477–1478. ACM, 2012.

[4] F. Brauer, R. Rieger, A. Mocan, and W. Barczynski. Enabling information extraction by inference of regular expressions from sample entities. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, pages 1285–1294. ACM, 2011.

[5] A. Cetinkaya. Regular expression generation through grammatical evolution. In *Proceedings of the 2007 GECCO conference*, GECCO '07, pages 2643–2646, New York, NY, USA, 2007. ACM.

[6] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *Evolutionary Computation, IEEE Transactions on*, 6(2):182 –197, apr 2002.

[7] B. Dunay, F. Petry, and B. Buckles. Regular language induction with genetic programming. In *Evolutionary Computation, 1994. IEEE World Congress on Computational Intelligence., Proceedings of the First IEEE Conference on*, volume 1, pages 396–400. IEEE, 1994.

[8] J. Friedl. *Mastering Regular Expressions*. O'Reilly Media, Inc., 2006.

[9] A. González-Pardo, D. Barrero, D. Camacho, and M. R-Moreno. A case study on grammatical-based representation for regular expression evolution. In Y. Demazeau, F. Dignum, J. Corchado, J. Bajo, R. Corchuelo, E. Corchado, F. Fernández-Riverola, V. Julián, P. Pawlewski, and A. Campbell, editors, *Trends in Practical Applications of Agents and Multiagent Systems*, volume 71 of *Advances in Intelligent and Soft Computing*, pages 379–386. Springer Berlin / Heidelberg, 2010.

[10] E. Kinber. Learning regular expressions from representative examples and membership queries. *Grammatical Inference: Theoretical Results and Applications*, pages 94–108, 2010.

[11] Y. Li, R. Krishnamurthy, S. Raghavan, S. Vaithyanathan, and A. Arbor. Regular Expression Learning for Information Extraction. *Computational Linguistics*, (October):21–30, 2008.

[12] E. Medvet and A. Bartoli. Brand-related events detection, classification and summarization on twitter. In *Proceedings of the 2011 IEEE/WIC/ACM International Conferences on Web Intelligence and Intelligent Agent Technology - Volume 01*, WI-IAT '12. IEEE Computer Society, 2012, to appear.

[13] R. Miller and A. Marshall. Cluster-based find and replace. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 57–64. ACM, 2004.

[14] R. Miller and B. Myers. Lapis: Smart editing with text structure. In *CHI'02 extended abstracts on Human factors in computing systems*, pages 496–497. ACM, 2002.

[15] R. Miller and B. Myers. Multiple selections in smart text editing. In *Proceedings of the 7th international conference on Intelligent user interfaces*, pages 103–110. ACM, 2002.

[16] R. Miller, B. Myers, et al. Lightweight structured text processing. In *Proceedings of 1999 USENIX Annual Technical Conference*, pages 131–144, 1999.

[17] E. Minkov, R. C. Wang, and W. W. Cohen. Extracting personal names from email: applying named entity recognition to informal text. In *Proceedings of the conference on Human Language Technology and Empirical Methods in Natural Language Processing*, HLT '05, pages 443–450, Stroudsburg, PA, USA, 2005. Association for Computational Linguistics.

[18] B. Svingen. Learning Regular Languages Using Genetic Programming. In J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, and R. Riolo, editors, *Genetic Programming 1998 Proceedings of the Third Annual Conference*, pages 374–376. Morgan Kaufmann, 1998.

[19] M. Tomita. Dynamic construction of finite automata from examples using hill-climbing. *Proceedings of the fourth annual cognitive science conference*, pages 105–108, 1982.

[20] T. Wu and W. Pottenger. A semi-supervised active learning algorithm for information extraction from textual data. *Journal of the American Society for Information Science and Technology*, 56(3):258–271, 2005.