Learning Text Patterns using Separate-and-Conquer Genetic Programming

Alberto Bartoli, Andrea De Lorenzo, Eric Medvet, and Fabiano Tarlao

DIA, Università degli Studi di Trieste, Italy
{bartoli.alberto,andrea.delorenzo,emedvet}@units.it
{fabiano.tarlao}@phd.units.it

Abstract. The problem of extracting knowledge from large volumes of unstructured textual information has become increasingly important. We consider the problem of extracting text slices that adhere to a syntactic pattern and propose an approach capable of generating the desired pattern automatically, from a few annotated examples. Our approach is based on Genetic Programming and generates extraction patterns in the form of regular expressions that may be input to existing engines without any post-processing. Key feature of our proposal is its ability of discovering automatically whether the extraction task may be solved by a single pattern, or rather a set of multiple patterns is required. We obtain this property by means of a separate-and-conquer strategy: once a candidate pattern provides adequate performance on a subset of the examples, the pattern is inserted into the set of final solutions and the evolutionary search continues on a smaller set of examples including only those not yet solved adequately. Our proposal outperforms an earlier state-of-the-art approach on three challenging datasets.

Keywords: Regular Expressions, Multiple Pattern, Programming by Example, Text Extraction

1 Introduction

The problem of extracting knowledge relevant for an end user from large volumes of unstructured textual information has become increasingly important over the recent years. This problem has many different facets and widely differing complexity levels, ranging from counting the number of occurrences of a certain word to extracting entities (e.g., persons and places) and semantics relations between them (e.g., lives-in). In this work, we are concerned with the *extraction* of text slices that adhere to a *syntactic pattern*. In particular, we investigate the feasibility of a framework where the pattern is to be generated automatically from a few *examples* of the desired extraction behavior provided by an end user.

A crucial difficulty involved in actually implementing a framework of this sort consists in generating a pattern that does not overfit the examples while at the same time providing high precision and recall on the full dataset to be processed. This difficulty is magnified when the syntactic features of the examples

are hardly captured adequately by a single pattern. For example, dates may be expressed in a myriad of different formats and learning a single pattern capable of expressing all these formats may be very difficult. Similarly, one might want to extract, e.g., dates and IP addresses, or URLs and Twitter hashtags. The learning machinery should be able to realize automatically, based on the expressiveness of the specific pattern formalism used, how many patterns are needed and then it should generate each of these patterns with an appropriate trade-off between specificity and generality.

In this paper we describe a system based on Genetic Programming that is capable of supporting a framework of this sort, by generating *automatically* text extractor patterns in the form of *regular expressions*. The user provides a text file containing a few text slices to be extracted, which have to be annotated, and the system automatically generates a set of regular expressions, where each element is specialized for a partition of the examples: processing a text stream with all these regular expressions will implement the desired extraction behavior. From an implementation point of view, our system actually generates a single regular expression composed of several regular expressions glued together by an OR operator. This choice allows using the generated expression with existing regex processing engines, e.g., those commonly used in Java or JavaScript, without any post-processing.

A key feature of our proposal is that the system does not need any hint from the user regarding the number of different patterns required for modelling the provided examples. Depending on the specific extraction task, thus, the system automatically discovers whether a single pattern suffices or a set of different patterns is required and, in this case, of which cardinality. We obtain this property by implementing a *separate-and-conquer* approach [1]. Once a candidate pattern provides adequate performance on a subset of the examples, the pattern is inserted into the set of final solutions and the evolutionary search continues on a smaller set of examples including only those not yet solved adequately. Of course, turning this idea into practice is difficult for a number of reasons, including the identification of suitable criteria for identifying the "adequate" level of performance.

We assess our proposal on several extraction tasks of practical complexity: dates expressed in many different formats to be extracted from bills enacted by the US Congress; URLs, Twitter citations, Twitter hashtags to be extracted from a corpus of Twitter posts; and IP addresses and dates expressed in different formats to be extracted from email headers. Our approach exhibits very good performance and significantly improves over a baseline constituted by an earlier proposal, the improvement being threefold: the generated patterns (i) exhibit better extraction precision and recall on unseen examples, (ii) are simpler, and (iii) are obtained with lower computational effort.

This work builds upon an earlier proposal for generating regular expressions automatically from annotated examples and counterexamples [2]. The cited work greatly improved over the existing state of the art for automatic generation of regular expressions for text processing [3–9]. However, the cited proposal was

designed for generating a *single* pattern capable of describing all the examples: as such, it is unable to effectively cope with scenarios requiring multiple patterns. The present work extends the cited proposal from a conceptual and from a practical point of view: the ability of discovering that multiple different patterns are required may greatly extend the scope of technologies for automatic pattern generation from examples.

Our separate-and-conquer approach bears several similarities to earlier approaches for rule induction, that aimed at synthesizing decision trees for solving classification problems [10, 11]: partial solutions with adequate performance on some partition are found with an evolutionary search; the data sample is recursively partitioned according to performance-related heuristics; and, the final solution is constructed by assembling the partial solutions. In fact, our approach might be modelled as a single design point amongst those that were analyzed by hyper-heuristic evolutionary search in the design space of rule induction for classification [11]. While such a point of view may be useful, it is important to remark that text classification and text extraction are quite different problems: the former may allow partitioning input units in two classes, depending on whether they contain relevant slices (e.g., [12–14]); the latter also requires identifying the boundaries of the slice—or *slices*—to be extracted.

The learning of text extractor patterns might be seen as a form of *programming by examples*, where a program in a given programming language is to be synthesized based on a set of input-output pairs. Notable results in this area have been obtained for problems of string manipulation solved by means of languages much richer than regular expressions [15–17]. The cited works differ from this proposal since (i) they output programs rather than regular expressions, (ii) they are tailored to fully specified problems, i.e., they do not need to worry about overfitting the data, and (iii) they exploit active learning, i.e., they assume an oracle exists which can mark extraction errors in order to improve the learning process.

2 Problem statement

A text *pattern* p is a predicate defined over strings: we say that a string s *matches* the pattern p if and only if $p(s)$ is true.

A *slice* x_s of a string s is a substring of s. A slice is identified by its starting and final indexes in the associated string. For ease of presentation, we will denote slices by their starting index and content, and we will specify the associated string implicitly. For instance, $\texttt{bana}_0$, $\texttt{na}_2$, $\texttt{an}_3$ and $\texttt{na}_4$ are all slices of the string `banana`. Slices of the same string are totally ordered by their starting index—e.g., $\texttt{na}_2$ precedes $\texttt{an}_3$. We say that x_s is a *superslice* of a slice x'_s (and x'_s is a *subslice* of x_s) if (i) x'_s is shorter than x_s, (ii) the starting index of x_s is smaller than or equal to the starting index of x'_s, and (iii) the final index of x_s is greater than or equal to the final index of x'_s—e.g., $\texttt{bana}_0$ is a superslice of $\texttt{na}_2$. We say that a slice x_s *overlaps* a slice x'_s of the same string if the intervals

of the indexes delimited by their starting and ending indexes have a non empty intersection—e.g., $\texttt{bana}_0$ overlaps $\texttt{nana}_2$.

An *extraction* of a set P of patterns in a string s is a slice x_s which meets the following conditions: (i) x_s matches a pattern in P, (ii) for each superslice x'_s of x_s, x'_s does not match any pattern in P, and (iii) for each other slice x'_s which overlaps x_s, either x_s precedes x'_s or x'_s does not match any pattern in P. We denote with $e(s, P)$ the set of all the extractions of P in s. For instance, let $s = \texttt{I␣said␣I␣wrote␣a␣ShortPaper}$ and P a set of only one pattern which describes (informally) "a word starting with a capital letter", then $e(s, P) = \{\texttt{I}_0, \texttt{I}_7, \texttt{ShortPaper}_{17}\}$. Note that the slices $\texttt{Short}_{17}$ and $\texttt{Paper}_{22}$ do not belong to $e(s, P)$, despite matching a pattern in P, as they do not meet the condition ii and iii above, respectively.

Finally, an *example* is a pair (s, X) where s is a string and X is a set of non-overlapping slices of s.

Based on the above definitions, the problem of learning a set of patterns from examples is defined as follows: given two sets of examples (E, E'), generate a set P of patterns using *only* E so that (i) the F-measure of P on E' is maximized and (ii) the complexity of P is minimized. The F-measure of P on E' is the harmonic mean of precision $\text{Prec}(P, E')$ and recall $\text{Rec}(P, E')$, which are defined as follows:

$$\text{Prec}(P, E') := \frac{\sum_{(s,X)\in E'} |e(s, P) \cap X|}{\sum_{(s,X)\in E'} |e(s, P)|} \tag{1}$$

$$\text{Rec}(P, E') := \frac{\sum_{(s,X)\in E'} |e(s, P) \cap X|}{\sum_{(s,X)\in E'} |X|} \tag{2}$$

The complexity of the set P of patterns depends on the formalism which is used to represents a pattern. In this work, we represent patterns by means of *regular expressions* and assume that the complexity of a regular expression is fully captured by its length. Hence, the complexity of P is given by $\ell(P) := \sum_{p\in P} \ell(p)$, where $\ell(p)$ is the length of the regular expression represented by p.

3 Our approach

We propose the use of Genetic Programming for solving the problem of learning a set of patterns—in the form of a set of regular expressions—from examples. An individual of the evolutionary search is a tree which represents a regular expression and we use common GP operators (crossover and mutation) in order to generate new individuals.

We learn a set of patterns according to a *separate-and-conquer* strategy, i.e, an iterative procedure in which, at each iteration, we learn a single pattern and then remove from the set of examples those which are "solved" by the learned pattern, repeated until no more examples remain "unsolved". At the end, the learned set of patterns is composed of the patterns learned at each iteration.

We describe the single evolutionary search (i.e., one iteration) in the next section and our separate-and-conquer strategy in Section 3.2.

3.1 Pattern evolutionary search

A pattern evolutionary search takes as input a *training set* $\mathcal{T}$ and outputs a single pattern p. The training set is composed of *annotated strings*, i.e., of tuples (s, X_d, X_u), where X_d and X_u are sets of non-overlapping slices of string s (i.e., no slice in X_d overlaps any slice in X_u). Slices in X_d are desired extractions of $\{p\}$ in s, whereas slices in X_u are undesired extractions of $\{p\}$ in s.

Our pattern evolutionary search is built upon the approaches proposed in [2, 12, 18], which we extend in three key aspects: (i) different fitness definitions (we use three objectives rather than two objectives); (ii) different fitness comparison criteria (we use a hierarchy between the fitness indexes rather than a Pareto-ranking); and, (iii) a mechanism for enforcing diversity among individuals.

An individual is a tree which represents a regular expression, i.e., a candidate pattern. The set of terminal nodes is composed of: (i) predefined ranges `a-z`, `A-Z` and `0-9`; (ii) character classes `\w` and `\d`; (iii) digits `0`, ..., `9`; (iv) partial ranges obtained from the slices in $\bigcup_{(s,X_d,X_u)\in\mathcal{T}} X_d$ according to the procedure described in [12]—a partial range being the largest interval of characters occurring in a set of strings (e.g., `a-c` and `l-n` are two partial ranges obtained from {`cabin`, `male`}), see the cited paper for full details); (v) other special characters such as `\.`, `:`, `@`, and so on. The set of function nodes is composed of: (i) the concatenator `••`; (ii) the character class `[•]` and negated character class `[^•]`; (iii) the possessive quantifiers `•*+`, `•++`, `•?+` and `•{•,•}+`; (iv) the non-capturing group `(?:•)`. A tree represents a string by means of a depth-first post order visit in which the • symbols in a non-terminal node are replaced by the string representations of its children.

The initialization of the population of n_{pop} individuals is based on the slices in $\bigcup_{(s,X_d,X_u)\in\mathcal{T}} X_d$, as follows (similarly to [18]). For each slice $x_s \in \bigcup_{(s,X_d,X_u)\in\mathcal{T}} X_d$, two individuals are built: one whose string representation is equal to the content of x_s where each digit is replaced by `\d` and each other alphabetic character is replaced by `\w`; another individual whose string representation is the same as the former and where consecutive occurrences of `\d` (or `\w`) are replaced by `\d++` (or `\w++`). For instance, the individuals `\d-\w\w\w-\d\d` and `\d-\w++-\d++` are built from the slice whose content is `7-Feb-2011`. If the number of individuals generated from $\mathcal{T}$ is greater than n_{pop}, exceeding individuals are removed randomly; otherwise, if it is lower than n_{pop}, missing individuals are generated randomly with a Ramped half-and-half method. Whenever an individual is generated whose string representation is not a valid regular expression, it is discarded and a new one is generated.

Each individual is a candidate pattern p and is associated, upon creation, with a *fitness* tuple $f(p) := (\text{Prec}(p, \mathcal{T}), \text{Acc}(p, \mathcal{T}), \ell(p))$—the first and second components are based on two operators $\sqcap$ and $\ominus$ defined over sets of slices as follows. Let X_1, X_2 be two sets of slices of the same string s. We define two operations between such sets. $X_1 \ominus X_2$ is the set of all the slices of s which (i) are a subslice of or equal to at least one slice in X_1, (ii) do not overlap any slice in X_2, and (iii) have not a superslice which meets the two first conditions. $X_1 \sqcap X_2$ is the set of all the slices of s which (i) are a subslice of or equal to at

least one slice in X_1, (ii) are a subslice of or equal to at least one slice in X_2, and (iii) have not a superslice which meets the two first conditions. For instance, let $X_1 = \{\mathtt{I}_0, \mathtt{I}_7, \mathtt{ShortPaper}_{17}\}$ and $X_2 = \{\mathtt{I}_0, \mathtt{Paper}_{22}\}$, then $X_1 \ominus X_2 = \{\mathtt{I}_7, \mathtt{Short}_{17}\}$, $X_1 \sqcap X_2 = \{\mathtt{I}_0, \mathtt{Paper}_{22}\}$.

The first fitness component is the precision on the annotated strings:

$$\mathrm{Prec}(p, \mathcal{T}) := \frac{\sum_{(s,X_d,X_u)\in\mathcal{T}} |e(s,\{p\}) \sqcap X_d|}{\sum_{(s,X_d,X_u)\in\mathcal{T}} |e(s,\{p\}) \sqcap (X_d \cup X_u)|} \tag{3}$$

The second component $\mathrm{Acc}(p, \mathcal{T})$ is the average of the True Positive Character Rate (TPCR) and True Negative Character Rate (TNCR):

$$\mathrm{TPCR}(p, \mathcal{T}) := \frac{\sum_{(s,X_d,X_u)\in\mathcal{T}} \|e(s,\{p\}) \sqcap X_d\|}{\sum_{(s,X_d,X_u)\in\mathcal{T}} \|X_d\|} \tag{4}$$

$$\mathrm{TNCR}(p, \mathcal{T}) := \frac{\sum_{(s,X_d,X_u)\in\mathcal{T}} \|(s \ominus e(s,\{p\}) \sqcap X_u\|}{\sum_{(s,X_d,X_u)\in\mathcal{T}} \|X_u\|} \tag{5}$$

where $\|X\|$ is the sum of the length of all the slices in X.

We compare individuals using a lexicographical order on their fitness tuples (also called *multi-layered fitness* [19]): between two individuals, the one with the greatest Prec is considered the best; in case they have the same Prec, the one with the greatest Acc is considered the best; in case, finally, they have the same Prec and Acc, the one with the lowest ℓ is considered the best. Figure 1 shows an example of the fitness of two individuals on an annotated string and shows which one is the best, according to the comparison criterion here defined.

$$\begin{aligned}
s &= \mathtt{10th_lap_lasted_from_7:02:11_to_11:10:13_of_02-03-79}\\
X_d &= \{\mathtt{7:02:11}_{21}, \mathtt{11:10:13}_{32}\}\\
X_u &= \{\mathtt{10th_lap_lasted_from}_{_0}, \mathtt{_of}_{_28}, \mathtt{_of}_{_40}\}\\
p_1 &= \mathtt{\backslash d\{2,2\}+.\backslash d\{2,2\}+.\backslash d\{2,4\}+}\\
p_2 &= \mathtt{(?:[0-9]++-?+)++)}\\
e(s,\{p_1\}) &= \{\mathtt{11:10:13}_{32}, \mathtt{02-03-79}_{44}\}\\
e(s,\{p_2\}) &= \{\mathtt{10}_0, \mathtt{7}_{21}, \mathtt{02}_{23}, \mathtt{11}_{26}, \mathtt{11}_{32}, \mathtt{10}_{35}, \mathtt{13}_{38}, \mathtt{02-03-79}_{44}\}\\
f(p_1) &= \left(1, 0.77 = \frac{1}{2}\left(\frac{8}{15} + \frac{29}{29}\right), 26\right)\\
f(p_2) &= \left(0, 0.83 = \frac{1}{2}\left(\frac{11}{15} + \frac{27}{29}\right), 17\right)
\end{aligned}$$

Fig. 1. Example of the fitness of two individuals p_1 and p_2 on an annotated string (s, X_d, X_u): according to our fitness comparison criterion, p_1 is better than p_2.

The population P is iteratively evolved as follows. At each iteration (generation), $0.1n_{\mathrm{pop}}$ new individuals are generated at random with a Ramped half-and-

half method, $0.1n_{\text{pop}}$ new individuals are generated by mutation and $0.8n_{\text{pop}}$ are generated by crossover. Mutation and crossover are the classic genetic operators applied to one or two individuals selected in P with a tournament selection: $n_{\text{tour}} = 7$ individuals are chosen at random in P and the best one is selected. Whenever an individual is generated whose string representation is the same as an existing individual, the former is discarded and a new one is generated—i.e., we enforce *diversity* among phenotypes. Among the resulting $2n_{\text{pop}}$ individuals, the best n_{pop} are chosen to form the new population. The procedure is stopped when either n_{gen} iterations have been executed or the fitness tuple of the best individual has remained unchanged for more than n_{stop} consecutive iterations.

The resulting pattern p is the one corresponding to the best individual at the end of the evolutionary search.

3.2 Separate-and-conquer strategy

We generate a set of patterns according to a separate-and-conquer strategy [1]. We execute an iterative procedure in which, at each iteration, we execute the pattern evolutionary search described in the previous section and then remove from the training set the slices correctly extracted by the set of patterns generated so far.

In order to avoid overfitting (i.e., in order to avoid generating a set P which performs well on E yet poorly on E'), we partition the set E of examples of the problem instance in two sets E_t and E_v. The partitioning is made just once, before executing the actual iterative procedure, and is made randomly so that the number of the slices in the training and validation sets are roughly the same, i.e., $\sum_{(s,X)\in E_t} |X| \approx \sum_{(s,X)\in E_v} |X|$. The training set E_t will be used by several independent executions of the iterative procedure, whereas E_v will be used (together with E_t) to assess the pattern sets obtained as outcomes of those executions and select just one pattern set as the final solution.

In detail the iterative procedure is as follows. Initially, let the set of patterns P be empty and let $\mathcal{T}$ include all the examples in the training set E_t: for each $(s, X) \in E_t$, a triplet $(s, X, \{s\} \ominus X)$ is added to $\mathcal{T}$ (i.e., $X_d := X$ and $X_u := \{s\} \ominus X$). Then, the following sequence of steps is repeated.

1. Apply an evolutionary search on $\mathcal{T}$ and obtain p.
2. If $\text{Prec}(p, \mathcal{T}) = 1$, then set $P := P \cup \{p\}$, otherwise terminate.
3. For each $(s, X_d, X_u) \in \mathcal{T}$, set $X_d := X_d \setminus e(s, \{p\})$;
4. If $\bigcup_{(s,X_d,X_u)\in\mathcal{T}} X_d$ is empty, then terminate.

In other words, at each iteration we aim at obtaining a pattern p with perfect precision (step 2). This pattern will thus extract only slices in X_d (i.e., slices which are indeed to be extracted) but it might miss some other slices. The next iterations will target the slices which are missed by p (step 3).

We insist on generating a pattern with perfect precision at each iteration because, if a pattern extracted something wrong, no other pattern could correct that error. As a consequence, we chose to use a multi-layered fitness during the

evolutionary search, where the most prominent objective is exactly to maximize $\mathrm{Prec}(p, \mathcal{T})$.

We execute the above procedure n_{job} independent times by varying the random seed (the starting set $\mathcal{T}$ remains the same) and we obtain n_{job} possibly different sets of patterns. At the end, we choose the one with the highest F-measure on $E = E_t \cup E_v$.

4 Experimental evaluation

We assessed our approach on three challenging datasets. *Bills* is composed of 600 examples where each string is a portion of a bill enacted by the US Congress and the slices corresponds to dates represented in several formats. *Tweets* is composed of 50 000 examples where each string is the text of a tweet (Twitter post) and the slices corresponds to URLs, Twitter citations and hashtags. Finally, *Headers* is composed of 101 examples where each string is the header of an email message and the slices corresponds to IP addresses and dates represented in several formats. We built the Bills dataset by crawling the web site of the US Congress and then applying a set of regular expressions to extract dates: we made this dataset available online[1] for easing comparative analysis. The Tweets and Headers datasets are derived from those used in [2]: the strings are the same but the slices are different. Table 1 shows 10 slices for each dataset, as a sample of the different formats involved.

Bills	Tweets	Headers
18.12.2013	@joshua␣seaton	10.236.182.42
2007/01/09	#annoyed	Thu,␣12␣Jan␣2012␣04:33:34␣-0800
23/03/2009	http://t.co/Bw7A5sbI	93.174.66.112
14-09-2011	#Anonymous	209.85.216.53
23,July␣2001	@YourAnonNews	24␣Jan␣2011␣09:36:00␣-0000
December␣31,␣2001	@zataz	27␣Apr␣2011␣09:31:01.0953
2000.01.27	@␣SweetDiccWilly	Mon␣Oct␣1␣13:04:58␣2012
Dec␣31,␣1991	http://t.co/bYxJ9NAE	Mon,␣01␣Oct␣2012␣12:05:40␣+0000
1997/12/31	#OpBlitzkrieg	151.76.78.168
1999-01-19	http://t.co/GrqKGECz	Mon,␣1␣Oct␣2012␣14:04:58␣+0200

Table 1. A sample of the slices in the the three datasets.

In order to obtain the slices from each string in a dataset, we manually built a set $P^\star$ of regular expressions which we then applied to the strings. Table 2 shows salient information about the datasets, including the number $|E \cup E'|$ of examples, the overall length $\sum_{(s,X)\in E\cup E'} \ell(s)$ of the strings, the overall number $\sum_{(s,X)\in E\cup E'} |X|$ of slices, the overall length $\sum_{(s,X)\in E\cup E'} \|X\|$ of the slices, and the number $|P^\star|$ of regular expressions used to extract the slices.

[1] http://regex.inginf.units.it/

	Examples		Slices		
Dataset	Number	Length	Number	Length	$\lvert P^\star \rvert$
Bills	600	16 510 800	3085	38 960	3
Tweets	50 000	4 344 275	71 621	933 646	2
Headers	101	261 174	1554	32 022	3

Table 2. Salient information about the datasets.

We built 15 different problem instances (E, E') for each dataset, by varying the overall number $\sum_{(s,X)\in E} |X|$ of slices in E and the random seed for partitioning the available examples in E and E'—25,50,100 slices, each obtained from 5 different seeds. Then, we applied our method to each problem instance and measured the F-measure of the generated P on E'. In order to provide a baseline for the results, we also applied the method proposed in [2]—which itself significantly improved over previous works on regular expression learning from examples—to the same problem instances. Since the cited method generates a *single* pattern p, for this method we set $P := \{p\}$.

We executed the experimental evaluation with the following parameter values: $n_{\text{pop}} = 500$, $n_{\text{gen}} = 1000$, $n_{\text{stop}} = 200$ and $n_{\text{job}} = 32$. We set the same values for the baseline, with the exception of n_{stop} which is not available in that method. We found, through an exploratory experimentation, that reasonable variations in these parameter did not alter the outcome of the comparison between our method and the baseline, which is summarized in Table 3.

	Num. of	Our method						Baseline					
Dataset	slices	Prec	Rec	Fm	$\ell(P)$	$\lvert P \rvert$	CE	Prec	Rec	Fm	$\ell(P)$	CE	ΔFm
Bills	25	0.47	0.60	0.49	56.4	3.2	2.3	0.22	0.51	0.24	26.4	2.5	104%
	50	0.59	0.69	0.62	76.6	4.0	6.9	0.27	0.51	0.27	97.2	6.9	129%
	100	0.68	0.81	0.73	88.6	4.6	11.3	0.41	0.52	0.39	104.6	11.6	87%
Tweets	25	0.99	0.92	0.94	24.6	2.4	0.6	0.90	0.86	0.87	25.6	1.1	8%
	50	0.97	0.98	0.96	22.4	2.6	1.6	0.86	0.88	0.85	27.2	2.1	13%
	100	0.98	0.99	0.99	25.6	3.0	3.2	0.85	0.96	0.90	46.2	4.1	10%
Headers	25	0.84	0.74	0.79	98.6	3.2	4.6	0.43	0.41	0.41	61.0	5.1	93%
	50	0.92	0.88	0.90	116.4	3.6	7.6	0.42	0.46	0.44	54.8	7.7	104%
	100	0.94	0.85	0.90	118.2	3.6	15.1	0.52	0.55	0.54	58.4	15.1	67%

Table 3. Results of the experimental evaluation. Computational Effort (CE) is expressed in 10^{10} character evaluations. $|P|$ is always equal to 1 for the baseline. ΔFm is the relative improvement of F-measure (in percentage) obtained by our method with respect to the baseline.

The most remarkable finding is the significant improvement of our method over the baseline, summarized in the rightmost column of Table 3 in the form of relative improvement of the F-measure on E' (we remark that E' are test data not available to the learning procedure). Indeed, raw results show that our

method obtained a greater F-measure in each of the 45 problem instances. The improvement is sharper for the Bills and Headers datasets (0.73 vs. 0.39 and 0.90 vs. 0.54, respectively, with 100 slices in the learning examples). These datasets exhibit a broad set of formats thus the ability of our approach to automatically discover the need of different patterns, as well as of actually generating them, does make a significant difference with respect to the baseline. Furthermore, there is an improvement also for the Twitter dataset, although the baseline exhibits very high F-measure for this dataset.

Another interesting finding concerns the complexity of the generated set of patterns. Though our method may generate, for a given problem instance, a set composed of more than one pattern, whereas the baseline always generates exactly one pattern, the overall complexity $\ell(P)$ is lower with our method in 5 on 9 problem instances. The difference is more noticeable for the Bills dataset. It is also important to remark that the average number of patterns discovered and generated by our method ($|P|$ column of Table 3) is close to the number of patterns used for annotating the dataset ($|P^\star|$ column of Table 2): our separate-and-conquer strategy does succeed in appropriately splitting the problem in several subproblems which can be solved with simpler patterns.

Table 3 shows also the *computational effort* (CE) averaged across problem instances with the same number of slices. We define CE as the number of character evaluations performed by individuals while processing a problem instance—e.g., a population of 100 individuals applied to a set E including strings totaling 1000 characters for 100 generations corresponds to $\mathrm{CE} = 10^7$. Note that this definition is independent of the specific hardware used. It can be seen that our method does not require a CE larger than the baseline. It can also be seen that for the Tweets dataset—the one for which the improvement in terms of F-measure and complexity of the solution was not remarkable—our method required a CE sensibly lower than the baseline. We think that this finding is motivated by our early termination criterion (determined by n_{stop}, see Section 3.1) which allows to spare some CE when no improvements are being observed during the evolutionary search.

Finally, we provide the execution time for the two methods with 25 slices, averaged over the 5 repetitions. Our method took 30 min, 3 min, and 29 min for Bills, Tweets, and Headers, respectively; the baseline took 45 min, 6 min, and 21 min, respectively. Each experiment has been executed on a machine powered with a 6 core Intel Xeon E5-2440 (2.40 GHz) equipped with 8 GB of RAM.

5 Concluding remarks

We considered the problem of learning a set of text extractor patterns from examples. We proposed a method for generating the patterns, in the form of regular expressions, which is based on Genetic Programming. Each individual represents a valid regular expression and individuals are evolved in order to meet three objectives: maximize the extraction precision on a training set, maximize the character accuracy on the training set, and minimize the regular expression

length (a proxy for its complexity). Several evolutionary searches are executed according to a iterative separate-and-conquer strategy: the examples which are "solved" at a given iteration are removed from the examples set of subsequent iterations. This strategy allows our method to automatically discover if several patterns are needed to solve a problem instance and, at the same time, to generate those patterns.

We assessed our method and compared its performance against an earlier state-of-the-art proposal. The experimental analysis, performed on several extraction tasks of practical complexity, showed that our method outperforms the baseline along three dimensions: greater extraction precision and recall on unseen examples, simpler patterns, and lower computational effort required to generate them.

References

1. Fürnkranz, J.: Separate-and-conquer rule learning. Artificial Intelligence Review **13**(1) (1999) 3–54
2. Bartoli, A., Davanzo, G., De Lorenzo, A., Medvet, E., Sorio, E.: Automatic synthesis of regular expressions from examples. Computer **47**(12) (Dec 2014) 72–80
3. Barrero, D.F., R-Moreno, M.D., Camacho, D.: Adapting searchy to extract data using evolved wrappers. Expert Systems with Applications **39**(3) (February 2012) 3061–3070
4. Brauer, F., Rieger, R., Mocan, A., Barczynski, W.: Enabling information extraction by inference of regular expressions from sample entities. In: ACM International Conference on Information and knowledge management, ACM (2011) 1285–1294
5. Kinber, E.: Learning regular expressions from representative examples and membership queries. Grammatical Inference: Theoretical Results and Applications (2010) 94–108
6. Barrero, D., Camacho, D., R-Moreno, M.: Automatic Web Data Extraction Based on Genetic Algorithms and Regular Expressions. Data Mining and Multi-agent Integration (2009) 143–154
7. Li, Y., Krishnamurthy, R., Raghavan, S., Vaithyanathan, S., Arbor, A.: Regular Expression Learning for Information Extraction. Computational Linguistics (October) (2008) 21–30
8. Cetinkaya, A.: Regular expression generation through grammatical evolution. In: International Conference on Genetic and evolutionary computation. GECCO, New York, NY, USA, ACM (2007) 2643–2646
9. Wu, T., Pottenger, W.: A semi-supervised active learning algorithm for information extraction from textual data. Journal of the American Society for Information Science and Technology **56**(3) (2005) 258–271
10. Barros, R.C., Basgalupp, M.P., de Carvalho, A.C., Freitas, A.A.: A hyper-heuristic evolutionary algorithm for automatically designing decision-tree algorithms. In: Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference, ACM (2012) 1237–1244
11. Pappa, G.L., Freitas, A.A.: Evolving rule induction algorithms with multi-objective grammar-based genetic programming. Knowledge and information systems **19**(3) (2009) 283–309

12. Bartoli, A., De Lorenzo, A., Medvet, E., Tarlao, F.: Playing regex golf with genetic programming. In: Proceedings of the 2014 conference on Genetic and evolutionary computation, ACM (2014) 1063–1070
13. Lucas, S.M., Reynolds, T.J.: Learning deterministic finite automata with a smart state labeling evolutionary algorithm. Pattern Analysis and Machine Intelligence, IEEE Transactions on **27**(7) (2005) 1063–1074
14. Lang, K.J., Pearlmutter, B.A., Price, R.A.: Results of the abbadingo one DFA learning competition and a new evidence-driven state merging algorithm. In: Grammatical Inference. Springer (1998) 1–12
15. Gulwani, S.: Automating string processing in spreadsheets using input-output examples. In: Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '11, New York, NY, USA, ACM (2011) 317–330
16. Menon, A., Tamuz, O., Gulwani, S., Lampson, B., Kalai, A.: A machine learning framework for programming by example. In: Proceedings of the 30th International Conference on Machine Learning (ICML-13). (2013) 187–95
17. Le, V., Gulwani, S.: Flashextract: A framework for data extraction by examples. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM (2014) 55
18. De Lorenzo, A., Medvet, E., Bartoli, A.: Automatic string replace by examples. In: Proceeding of the fifteenth annual conference on Genetic and evolutionary computation conference, ACM (2013) 1253–1260
19. Eggermont, J., Kok, J.N., Kosters, W.A.: Genetic programming for data classification: Partitioning the search space. In: Proceedings of the 2004 ACM symposium on Applied computing, ACM (2004) 1001–1005