Inference of Regular Expressions for Text Extraction from Examples

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

Abstract—A large class of entity extraction tasks from text that is either semistructured or fully unstructured may be addressed by regular expressions, because in many practical cases the relevant entities follow an underlying syntactical pattern and this pattern may be described by a regular expression. In this work we consider the long-standing problem of synthesizing such expressions automatically, based solely on examples of the desired behavior.
We present the design and implementation of a system capable of addressing extraction tasks of realistic complexity. Our system is based on an evolutionary procedure carefully tailored to the specific needs of regular expression generation by examples. The procedure executes a search driven by a multiobjective optimization strategy aimed at simultaneously improving multiple performance indexes of candidate solutions while at the same time ensuring an adequate exploration of the huge solution space.
We assess our proposal experimentally in great depth, on a number of challenging datasets. The accuracy of the obtained solutions seems to be adequate for practical usage and improves over earlier proposals significantly. Most importantly, our results are highly competitive even with respect to human operators. A prototype is available as a web application at http://regex.inginf.units.it.

Index Terms—Genetic Programming, Information extraction, Programming by examples, Multiobjective optimization, Heuristic search

1 Introduction

A REGULAR EXPRESSION is a means for specifying string patterns concisely. Such a specification may be used by a specialized engine for extracting the strings matching the specification from a data stream. Regular expressions are a long-established technique for a large variety of application domains, including text processing, and continue to be a routinely used tool due to their expressiveness and flexibility. A large class of entity *extraction* tasks, in particular, may be addressed by regular expressions, because in many practical cases the relevant entities follow an underlying syntactical pattern and this pattern may be described by a regular expression. However, the construction of regular expressions capable of guaranteeing high precision and high recall for a given extraction task is tedious, difficult and requires specific technical skills.

In this work, we consider the problem of synthesizing a regular expression *automatically*, based solely on *examples* of the desired behavior. This problem has attracted considerable interest, since a long time and from different research communities. A wealth of research efforts considered *classification* problems in *formal* languages [1], [2], [3], [4], [5], [6]—those results are not immediately useful for text extraction. Essentially, the problem considered by those efforts consisted in inferring an acceptor for a regular language based on positive and negative sample strings, i.e., of strings described by the language and of strings not described by the language. Learning of *deterministic finite automata (DFA)* from examples was also a very active area, especially because of competitions that resulted in several important insights and algorithms, e.g. [7], [8]. Such research, however, usually considered problems that were not inspired by any real world application [8] and the applicability of the corresponding learning algorithms to other application domains is still largely unexplored [9]. For example, the so-called Abbadingo competition was highly influential in this area and considered short sequences of binary symbols, with training data drawn uniformly from the input space. Settings of this sort do not fit the needs of practical text processing applications, which have to cope with much longer sequences of symbols, from a much larger alphabet, not drawn uniformly from the space of all possible sequences. Furthermore, regular expressions used in modern programming languages allow specifying more various extraction tasks than those which can be specified using a DFA.

A text extraction problem was addressed by researchers from IBM Almaden and the University of Michigan, which developed a procedure for improving an initial regular expression to be provided by the user based on examples of the desired functioning [10]. The cited work is perhaps the first one addressing entity extraction from real text of non trivial size and complexity: the entities to be extracted included software names, email addresses and phone numbers while the datasets were unstructured and composed of many thousands of lines. A later proposal by researchers from IBM India and Chennai Mathematical Institute still required an initial regular expression but was more robust toward initial expressions of modest accuracy and noisy datasets [11]. Refinement of a given regular expression was also considered by an IBM Research group, which advocated involvement of a human operator for providing feedback during the process [12]. The need of an initial solution was removed by researchers from SAP AG that demonstrated the practical feasibility of inferring a regular expression from scratch, based solely on a set of examples

• *All the authors are with the Department of Engineering and Architecture (DIA), University of Trieste, Italy.*
E-mail: bartoli.alberto@units.it.

Manuscript received . . .

derived from enterprise data, such as, e.g., a product catalog or historical invoices [13]. A more recent proposal of ours has obtained further significant improvements in this area, in terms of precision and recall of the generated solutions as well as in terms of smaller amount of training data required [14], [15]. Regular expressions for text extraction tasks of practical complexity may now be obtained in a few minutes, based solely on a few tens of examples of the desired behavior.

In this work we present a system that aims at improving the state-of-the-art in this area. Our proposal is internally based on *Genetic Programming (GP)*, an evolutionary computing paradigm which implements a heuristic search in a space of candidate solutions [16]. We execute a search driven by a *multiobjective optimization* strategy aimed at simultaneously improving multiple performance indexes of candidate solutions while at the same time ensuring an adequate exploration of the huge solution space. Our proposal is a significant improvement and redesign of the approach in [15], resulting in a system that generates solutions of much better accuracy. The improvements include: (a) a radically different way of quantifying the quality of candidate solutions; (b) inclusion, in the starting points of the search, of candidate solutions built based on an analysis of the training data, rather than being fully random; (c) a strategy for restricting the solution space by defining potentially useful "building blocks" based on an analysis of the training data; and (d) a simple mechanism for enforcing structural diversity of candidate solutions.

Furthermore, the redesign features several novel properties which greatly broaden the scope of extraction tasks that may be addressed effectively:

- *Support for the or operator*. In many cases learning a single pattern capable of describing all the entities to be extracted may be very difficult—e.g., dates may be expressed in a myriad of different formats. Our system is able to address such scenarios by generating several regular expressions that are all joined together with *or* operators to form a single, larger regular expression. We implement this functionality by means of a *separate-and-conquer* procedure [17], [18], [19]. Once a candidate regular expression provides adequate accuracy on a subset of the examples, the expression is inserted into the set of final solutions and the learning process continues on a smaller set of examples including only those not yet solved adequately [20]. The key point is that the system is able to realize automatically how many regular expressions are needed.
- *Context-dependent extraction*. It is often the case that a text snippet must or must not be extracted depending on the text surrounding the snippet—e.g., an email address might have to be extracted only when following a `Reply-To:` header name. Modern regular expression engines provide several constructs for addressing these needs but actually taking advantage of those constructs is very challenging: the more the available constructs, the larger the search space. Our system is able to generate regular expressions which exploit *lookaround* operators effectively, i.e., operators specifying constraints on the text that precedes or follows the text to be extracted.
- *No constraints on the size of training examples*. We place no constraints on the size of training examples: the training data may consist of either a single, potentially very large, file with an annotation of all the desired extractions, or of a set of lines with zero or more extractions in each one. This seemingly minor detail may in fact be quite important in practice: the cited work [15] was not able to exploit training examples including multiple extractions correctly (this point will be discussed in detail later), thus the training data had to be segmented in units containing at most one extraction and in such a way that desired extractions did not span across adjacent units. The need for such a tricky operation is now removed. Accommodating the possibility of multiple extractions in each training example has required significant changes in the search strategy internally used by the system.

We assess our proposal experimentally in great depth, on a number of challenging datasets of realistic complexity and with a very small portion of the dataset available for learning. We compare precision and recall of the regular expressions generated by our system to significant baseline methods proposed earlier in the literature. The results indicate a clear superiority of our proposal and the obtained accuracy values seem to be adequate for practical usage. Our results are highly competitive also with respect to a pool of more than 70 human operators, both in terms of accuracy and of time required for building a regular expression. Indeed, we are not aware of any proposal for automatic generation of regular expressions in which human operators were used as a baseline.

We made publicly available the source code of our system (https://github.com/MaLeLabTs/RegexGenerator) and deployed an implementation as a web app (http://regex.inginf.units.it).

2 Related work

In this section we discuss further proposals that, beyond those already discussed in the introduction, may be useful to place our work in perspective with respect to the existing literature. As pointed out by [10], the learning of regular expressions for information extraction prior to the cited work focused on scenarios characterized by alphabet sizes much smaller than those found in natural language text. Rather than attempting to infer patterns over the text to be extracted, the usual approach consisted on learning patterns over *tokens* generated with various text processing techniques, e.g., POS tagging, morphological analysis, gazetteer matching [21], [22], [23].

An attempt at learning regular expressions over real text was proposed in [24]. The cited work considered reduced forms of regular expressions (a small subset of POSIX rules) and, most importantly, considered a simple classification problem consisting in the detection of HTML lines with a link to other web documents. Text classification and text extraction are related but different problems, though. The former assumes an input stream segmented in units to be processed one at a time; one has to detect whether the given input unit contains at least one interesting substring. The latter requires instead the ability to identify, in the (possibly very long) input stream, the boundaries of all

the relevant substrings, if any. Furthermore, text extraction usually requires the ability to identify a context for the desired extraction, that is, a given sequence of characters may or may not have to be extracted depending on its surroundings. Interestingly, the approach in [15] was developed for extraction but delivered better results than in [24] also in classification.

Further proposals for addressing classification problems have been developed but tailored to very specific scenarios, recent examples include email spam campaigns [25], [26] and clinical symptoms [27].

There have been other proposals for regular expression learning aimed at information extraction from real text, specifically web documents [28]. The cited work provides an accuracy in URL extraction from real web documents that is quite low—the reported value for F-measure being 27% (on datasets that are not public). In this respect, it is useful to observe that the latest proposal [15] obtained accuracy well above 90% in the 12 datasets considered; moreover, two of those datasets were used also in [10], [13] and in those cases it obtained similar or much better accuracy with a training set smaller by an order of magnitude.

The problem of learning a regular expression by examples of the desired extraction behavior could be seen as a very specific problem in the broader category of *programming by examples*, where a program in a given programming language is to be synthesized based on a set of input-output pairs [29]. In particular, the problem is an under-specified task [30] in the sense that there may usually be many different solutions whose behavior on the training data is identical while their behavior on unseen data is different. The cited work considers the generation of regular expressions for classification tasks on phone numbers, dates, email addresses and URLs—tasks that are considered to be tricky even for expert developers and to lack an easy-to-formalize specification. It advocates the writing of solutions by several expert developers based on some examples, an assessment of their behavior on unseen data made in *crowd-sourcing*, and an evolutionary optimization of the available solutions based on the feedback from the crowd. Our proposal generates a regular expression in a fully automatic way. Furthermore, we assess our work on datasets that are orders of magnitude larger than those considered in [30] and on tasks that seems fair to define much more challenging. Of course, we make these observations in the attempt of clarifying our proposal and by no means we intend to criticize the cited work: besides, the cited work investigates the possibility of crowd-sourcing difficult programming tasks and is *not* meant to propose a method for the automatic generation of regular expressions from examples. It is useful to observe, though, that the authors of the cited work were not aware of any approach suitable for learning regular expressions capable of handling the large alphabet sizes occurring in real-world text files, while such functionality was demonstrated in [13], [14], [15].

As pointed out above, learning a program from examples of the desired behavior is an intrinsically under-specified task—there might be many different solutions with identical behavior over the examples. Furthermore, in practice, there is usually not even any guarantee that a solution which perfectly fits all the examples actually exists. The common approach for addressing this issue, which is also our approach, aims at an heuristic balance between generalization and overfitting: we attempt to infer from the examples what is the actual desired behavior, without insisting on obtaining perfect accuracy on the training set. It may be worth mentioning that coding challenges exist (and occasionally become quite popular in programming forums) which are instead aimed at overfitting a list of examples [31], [32]. The challenge[1] consists in writing the shortest regular expression that matches all strings in a given list and does not match any string in another given list. Our proposal is not meant to address these scenarios. From the point of view of our discussion, scenarios of this sort differ from text extraction in several crucial ways. First, they are a classification problem rather than an extraction problem. Second, they place no requirements on how strings not listed in the problem specification should be classified—e.g., strings in the problem specification followed or preceded by additional characters. Text extraction requires instead a form of generalization, i.e., the ability of inducing a general pattern from the provided examples.

Finally, we mention a recent proposal for information extraction from examples [33]. The cited work describes a powerful and sophisticated framework for extracting multiple different fields automatically in semi-structured documents. As such, the framework encompasses a much broader scenario than our work. A tool implementing this framework has been publicly released as part of Windows Powershell[2]. The tool does not generate a regular expression; instead, it generates a program in a specified algebra of string processing operators that is to be executed by a dedicated engine. We decided to include this tool in our experimental evaluation in order to obtain further insights into our results.

3 Scenario

We are concerned with the task of generating a regular expression which can generalize the extraction behavior represented by some examples, i.e., by strings annotated with the desired portions to be extracted. In this section we define the problem statement in detail along with the notation which will be used hereafter.

We focus on the regular expression implementation which is provided by the Java standard libraries. A deep comparison of different flavours of regular expressions is beyond the scope of this paper [34], yet it is worth to mention that Java regular expressions provide more constructs than POSIX extended regular expressions (ERE)—e.g., lookarounds (see Section 4.1.1)—which allow to define patterns in a more compact form.

3.1 Definitions

A *snippet* x_s of a string s is a substring of s, identified by the starting and ending index in s. For readability, we refer to snippets using their textual content followed by their starting index as subscript—e.g., ex_5, extra_5 and $\mathsf{traction}_7$, are three different snippets of the string $\mathsf{text\_extraction}$. We denote

1. https://www.google.it/search?q=regex+golf
2. Windows Management Framework 5.0 Preview, November 2014.

by $\mathcal{X}_s$ the set of all the snippets of s. Let $x_s, x'_s \in \mathcal{X}_s$. A total order is defined among snippets in $\mathcal{X}_s$ based on their starting index: x_s *precedes* x'_s if the starting index of the former is strictly lower than the starting index of the latter. We say that x_s is a *supersnippet* of x'_s if the indexes interval of x_s strictly contains the indexes interval of x'_s: in this case, x'_s is a *subsnippet* of x_s. Finally, we say that x_s *overlaps* x'_s if the intersection of their index intervals is not empty. For instance, ex_1, ex_5, extra_5 and $\mathsf{traction}_7$, are snippets of the string text␣extraction: extra_5 is a supersnippet of ex_5 (but not of ex_1), extra_5 precedes and overlaps $\mathsf{traction}_7$.

A regular expression r applied on a string s deterministically extracts zero, one or more snippets. We denote the (possibly empty) set of such snippets, that we call *extractions*, by $[\mathcal{X}_s]_r$.

3.2 Problem statement

The problem input consists of a set of *examples*, where an example (s, X_s) is a string s associated with a (possibly empty) set of non-overlapping snippets $X_s \subset \mathcal{X}_s$. String s may be, e.g., a text line, or an email message, or a log file and so on. Set X_s represents the *desired extractions* from s, whereas snippets in $\mathcal{X}_s \setminus X_s$ are *not* to be extracted.

Intuitively, the problem consists in learning a regular expression $\hat{r}$ whose extraction behavior is consistent with the provided examples—$\hat{r}$ should extract from each string s only the desired extractions X_s. Furthermore, $\hat{r}$ should capture the *pattern* describing the extractions, thereby *generalizing* beyond the provided examples. In other words, the examples constitute an incomplete specification of the extraction behavior of an ideal and unknown regular expression $r^\star$. The learning algorithm should aim at inferring the extraction behavior of $r^\star$ rather than merely obtaining from the example strings exactly the desired extractions. We formalize this intuition as follows.

Let E and $E^\star$ be two different sets of examples, both representing the extraction behavior of a target regular expression $r^\star$. The problem consists in learning, from *only* the examples in E, a regular expression $\hat{r}$ which maximizes its F-measure on $E^\star$, i.e., the harmonic mean of precision and recall w.r.t. the desired extractions from the examples in $E^\star$:

$$\mathrm{Prec}(\hat{r}, E^\star) := \frac{\sum_{(s,X_s)\in E^\star} |[\mathcal{X}_s]_{\hat{r}} \cap X_s|}{\sum_{(s,X_s)\in E^\star} |[\mathcal{X}_s]_{\hat{r}}|}$$

$$\mathrm{Rec}(\hat{r}, E^\star) := \frac{\sum_{(s,X_s)\in E^\star} |[\mathcal{X}_s]_{\hat{r}} \cap X_s|}{\sum_{(s,X_s)\in E^\star} |X_s|}$$

The greater the F-measure of $\hat{r}$ on $E^\star$, the more similar the extraction behaviour of $\hat{r}$ and $r^\star$.

We call the pair of sets of examples $(E, E^\star)$ a *problem instance*. In our experimental evaluation we built several problem instances starting from quite complex target expressions $r^\star$ and strings consisting of real world datasets (e.g., logs, HTML lines, Twitter posts, and alike). Of course, in a practical deployment of the system set $E^\star$ is not available because the target expression $r^\star$ is not known.

3.2.1 Observations on the problem statement

We point out that characterizing the features of a problem instance which may impact the quality of a generated solution is beyond the scope of this paper. Assessing the difficulty of a given problem instance, either in general or when solved by a specific approach, is an important theoretical and practical problem. Several communities have long started addressing this specific issue, e.g., in information retrieval [35], [36] or in pattern classification [37], [38]. Obtaining practically useful indications, though, is still a largely open problem, in particular, in evolutionary computing [39] as well as in more general search heuristics [40], [41].

A notable class of problem instances is the one which we call *with context*. Intuitively, these are the problem instances in which a given sequence of characters is the textual content of snippet to be extracted and also the textual content of a snippet which is not to be extracted. For example, consider a problem instance with the two examples $(\mathsf{I␣have␣12␣dogs}, \emptyset)$ and $(\mathsf{Today␣is␣7\text{-}12\text{-}11}, \{\mathsf{12}_{11}\})$. This problem instance is with context because the sequence of characters 12 is not to be extracted from the first example but is to be extracted from the second example. The discriminant between the two cases is in the portion of the string surrounding the sequence 12, that is, in its context. Of course, similar scenarios could occur with respect to sequences of characters in the same example rather than in different examples—e.g., assuming an email message is an example, one might want to extract only the email addresses following a Reply-To: header name.

4 Our approach

Our approach is based on *Genetic Programming* (GP) [16]. GP is an evolutionary computing paradigm in which candidate solutions for a target problem, called *individuals*, are encoded as trees. A problem-dependent numerical function, called *fitness*, must be defined in order to quantify the ability of each individual to solve the target problem. This function is usually implemented by computing a performance index of the individual on a predefined set of problem instances, called the *learning set*. A GP execution consists of an heuristic and stochastic search in the solution space, looking for a solution with optimal fitness. To this end, an initial population of individuals is built, usually at random, and an iterative procedure is performed which consists in (i) building new individuals from existing ones using genetic operators (usually *crossover* and *mutation*), (ii) adding new individuals to the population, and (iii) discarding worst individuals. The procedure is repeated a predefined number of times or until a predefined condition is met (e.g., a solution with perfect fitness is found).

We carefully adapted the general framework outlined above to the specific problem of regular expression generation from examples. Our GP procedure is built upon our earlier proposal [15]—the numerous improvements were listed in the introduction. We describe this procedure in detail in the next sections: encoding of regular expressions as trees (Section 4.1.1), fitness definition (Section 4.1.2), construction of the initial population and its evolution for exploring the solution space (Section 4.1.3). Next, we describe our separate-and-conquer strategy (Section 4.1.4) and the overall organization of GP searches (Section 4.2).

4.1 GP search

We designed a *GP search* which takes a *training set* $\mathcal{T}$ as input and outputs a regular expression $\hat{r}$. The training set is composed of tuples (s, X_s^d, X_s^u), the components of each tuple being as follows: (i) a string s; (ii) a set of snippets X_s^d representing the desired extractions from s; (iii) a set of snippets X_s^u representing the undesired extractions from s, i.e., no snippet of s overlapping a snippet in X_s^u should be extracted. The training set $\mathcal{T}$ must be constructed such that $\forall s \in \mathcal{T}$ (i) $X_s^d \cap X_s^u = \emptyset$, and, (ii) snippets in $X_s^d \cup X_s^u$ must not overlap each other. The goal of a GP search is to generate a regular expression r such that $\forall s \in \mathcal{T}, X_s^d = [\mathcal{X}_s]_r$. We recall that, from a broader point of view, the generated regular expression r should generalize beyond the examples in $\mathcal{T}$ (see Section 3.2).

4.1.1 Tree representation

In our proposal an individual is a tree which represents a regular expression r. Each node in a tree is associated with a *label*, which is a string representing basic components of a regular expressions that are available to the GP search (discussed in detail below). Labels of non-leaf nodes include the placeholder symbol $\divideontimes$: each children of a node is associated with an occurrence of symbol $\divideontimes$ in the label of that node. The regular expression represented by a tree is the string constructed by means of a depth-first post-order visit of the tree. In detail, we execute a *string transformation* of the root node of that tree. The string transformation of a node is a string obtained from the node label where each $\divideontimes$ symbol is replaced by the string transformation of the associated child. Figure 1 shows two examples of tree representations of regular expressions.

Available labels are divided in two sets: a set of predefined labels which represent regular expression constructs, and a set of $\mathcal{T}$-dependent labels constructed as described below. In other words, the GP search explores a space composed of candidate solutions assembled from general regular expression constructs and from components constructed before starting the GP search by analyzing the provided examples—this procedure was not present [15]. The rationale for $\mathcal{T}$-dependent labels consists in attempting to shrink the size of the solution space by identifying those sequences of characters which occurs often in the desired extractions (or "around" them) and making these sequences available to the GP search as unbreakable building blocks. For instance, in the task of generating a regular expression for extracting URLs, the string `http` could be an useful such block.

Predefined labels are the following: character classes (`\d`, `\w`), predefined ranges (`a-z`, `A-Z`), digits (`0`, ..., `1`), predefined characters (`\„`, `:`, `„`, `;`, `_`, `=`, `"`, `'`, `\\`, `/`, `\?`, `\!`, `\}`, `\{`, `\(`, `\)`, `\[`, `\]`, `<`, `>`, `@`, `#`, `␣`), concatenator ($\divideontimes\divideontimes$), set of (un)possible matches ([$\divideontimes$], [^$\divideontimes$]), possessive quantifiers ($\divideontimes$`*+`, $\divideontimes$`++`, $\divideontimes$`?+`, $\divideontimes\{\divideontimes,\divideontimes\}$`+`), non-capturing group ((?:$\divideontimes$)), and lookarounds ((?<=$\divideontimes$), (?<!$\divideontimes$), (?=$\divideontimes$), (?!$\divideontimes$)). We include possessive quantifiers and we do not include greedy and lazy quantifiers[3] because greedy and lazy quantifiers have worst-case exponential complexity, which results in execution times for fitness evaluation too long to be practical [15]. We include lookarounds for addressing problem instances with context, i.e., scenarios where a given sequence of characters has or has not to be extracted depending on its surroundings (Section 3.2.1)—lookarounds were not used in [15]. Lookaround is a shorthand for regular expression constructs which allow defining constraints on the text that either precedes or follows the snippet to be extracted, in the form of text that must or must not be present (see [34] for details). For instance, the regular expression $r =$ `(?<=\d\d-\d\d-)\d++` contains a lookaround operator, the *positive lookbehind* operator, that specifies which text must precede the snippet to be extracted. Given the string $s =$ `born:␣02-03-1979,␣graduated:␣21-07-04,␣age:␣35`, the set of extractions $[\mathcal{X}_s]_r$ contains $\texttt{1979}_{12}$ and $\texttt{04}_{37}$, but not $\texttt{35}_{44}$. Some notable regular expression implementations (namely JavaScript) does not work with lookbehind.

The set of $\mathcal{T}$-dependent labels contains *token* labels and *partial range* labels.

Token labels are generated as follows. A multiset T^d of candidate tokens is built by applying the regular expression `\w+|\s+|[^\w\s]+` to each desired extraction in $\mathcal{T}$: that is, T^d contains all the extractions obtained by that regular expression on each element of $\bigcup_{(s,X_s^d,X_s^u)\in\mathcal{T}} X_s^d$. Then, the occurrency rate of each candidate token is computed as its multiplicity in T^d divided by $|\bigcup_{(s,X_s^d,X_s^u)\in\mathcal{T}} X_s^d|$. Finally, candidate tokens with an occurrency rate which is greater than 80% are retained as token labels. The same procedure is executed with respect to candidate tokens obtained from undesired extractions (only in tuples (s, X_s^d, X_s^u) for which $X_s^d \neq \emptyset$).

Partial range labels are obtained as the largest intervals of alphanumeric characters whose elements occur in the desired extractions (i.e., in $\bigcup_{(s,X_s^d,X_s^u)\in\mathcal{T}} X_s^d$). For instance, `a-c` and `l-n` are two partial ranges labels obtained from the strings `cabin` and `male`.

4.1.2 Fitness

The fitness definition, i.e., how to quantify the quality of a candidate solution for the problem being solved, is a fundamental design decision in GP. Several practical applications are based on a *multiobjective* approach, where the quality of a candidate solution is assessed by means of *two* fitness indexes: one for quantifying performance, the other for quantifying a complexity index of the solution, typically its size. Such an approach has proven to be very effective at preventing *bloat*, i.e., the proliferation of candidate solutions that grow bigger in size without any corresponding improvement in performance [42].

We developed a fitness definition in which the performance of the solution is taken into account by *two* performance indexes (differently from the single one used in [15]): one considers examples at the level of full extractions; the other considers instead each example as a character sequence where each character, specified by its value *and* position, is to be classified between extracted vs. non extracted. The aim of the latter is to rewards small improvements at the character level in the extraction behavior, even when they do not result in new full snippets correctly extracted. The same aim motivated the fitness definition of [15], but here we accomodate a scenario with multiple extractions for each example, which was not tailored by the cited paper.

3. Greedy quantifiers: $\divideontimes$* $\divideontimes$+ $\divideontimes$? $\divideontimes\{\divideontimes,\divideontimes\}$. Lazy quantifiers: $\divideontimes$*? $\divideontimes$+? $\divideontimes$?? $\divideontimes\{\divideontimes,\divideontimes\}$?

We couple the two performance indexes with length of the regular expression, thereby resulting in three fitness indexes.

Our fitness definition requires comparing the actual extractions generated by a regular expression to the desired extractions. To this end, we define two operators over sets of snippets. Let X_s and X'_s be two sets of snippets of s. The *snippet set difference* $X_s \ominus X'_s$ is the set composed of each snippet in $\mathcal{X}_s$ which satisfies the following conditions: (i) is a subsnippet of, or is equal to, one or more snippets in X_s, (ii) does not overlap any snippet in X'_s, (iii) is not a subsnippet of any snippet which meets the two previous conditions. For instance, consider string $s = \texttt{I␣said␣I␣wrote␣a␣ShortPaper}$ and the sets of snippets $X_s = \{\texttt{I}_0, \texttt{I}_7, \texttt{ShortPaper}_{17}\}$, and $X'_s = \{\texttt{I}_0, \texttt{Pap}_{22}\}$. It will be $X_s \ominus X'_s = \{\texttt{I}_7, \texttt{Short}_{17}, \texttt{er}_{25}\}$. The *snippet set intersection* $X_s \sqcap X'_s$ is defined in the same way except that condition ii requires to be a subsnippet of, or to be equal to, one or more snippets in X'_s. In the previous example it will be $X_s \sqcap X'_s = \{\texttt{I}_0, \texttt{Pap}_{22}\}$.

Each individual r is associated with a fitness tuple $f(r) := (\text{Prec}(r, \mathcal{T}), \text{Acc}(r, \mathcal{T}), \ell(r))$. The first component $\text{Prec}(r, \mathcal{T})$ of the fitness is the precision on the tuples in $\mathcal{T}$:

$$\text{Prec}(r, \mathcal{T}) := \frac{\sum_{(s, X_s^d, X_s^u) \in \mathcal{T}} \left| [\mathcal{X}_s]_r \sqcap X_s^d \right|}{\sum_{(s, X_s^d, X_s^u) \in \mathcal{T}} \left| [\mathcal{X}_s]_r \sqcap (X_s^d \cup X_s^u) \right|}$$

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

$$\text{TPCR}(r, \mathcal{T}) := \frac{\sum_{(s, X_s^d, X_s^u) \in \mathcal{T}} \left\| [\mathcal{X}_s]_r \sqcap X_s^d \right\|}{\sum_{(s, X_s^d, X_s^u) \in \mathcal{T}} \| X_s^d \|}$$

$$\text{TNCR}(r, \mathcal{T}) := \frac{\sum_{(s, X_s^d, X_s^u) \in \mathcal{T}} \left\| (\{s_0\} \ominus [\mathcal{X}_s]_r) \sqcap X_s^u \right\|}{\sum_{(s, X_s^d, X_s^u) \in \mathcal{T}} \| X_s^u \|}$$

where $\|X\|$ is the sum of the length of all the snippets in X and s_0 is the snippet consisting of the whole string s.

Finally, the latter component $\ell(r)$ is the length of the regular expression r (this index has to be minimized, unlike the other two indexes which have to be maximized).

We rank individuals based on their fitness tuples as follows. An individual *a Pareto-dominates* another individual b if a is better than b on at least one fitness element and not worse on the other elements. An individual belongs to the ith *frontier* if and only if it is Pareto-dominated only by individuals belonging to jth frontier, with $j < i$ (individuals in the first frontier are not Pareto-dominated by any other individual). Based on these definitions, we first sort individuals based on the Pareto frontier they belong to. Second, we establish a total order among individuals belonging to the same Pareto frontier based on a lexicographic ordering among fitness indexes.

4.1.3 Initialization and evolution

Our GP search operates on a fixed-size population of n_{pop} individuals. We build the initial population basing on the training set $\mathcal{T}$, unlike the usual approach in GP which consists of building the entire population at random (as in [15]). We generate 4 individuals from each snippet in each example, all generated so as to extract that snippet. The rationale is to provide a sort of good starting point and useful genetic material for the search.

In detail, for each snippet x_s in $\bigcup_{(s, X_s^d, X_s^u) \in \mathcal{T}} X_s^d$, we generate 4 individuals as described below—Figure 1 shows an example of the procedure applied to a single snippet.

a) A tree is generated from the textual content of x_s using, whenever possible and with decreasing priority, (i) nodes with token label to represent the corresponding tokens, (ii) nodes with the label \d to represent digits, (iii) subtrees corresponding to [a-zA-Z] to represent alphabetic characters, (iv) nodes with predefined characters labels to represent corresponding characters, and (v) nodes with the label . for all other characters.
b) A tree is generated as in a, then subtrees composed only of nodes with two labels, one being the concatenator ⚹⚹ and the other a generic label l, are replaced by the subtree corresponding to l++.
c) Two snippets x_s^{behind} and x_s^{ahead} are considered such that their length is at most $10\ell(x_s)$ and they immediately precede (x_s^{behind}) or succeed (x_s^{ahead}) x_s in the corresponding tuple—they are not considered if x_s stays at the beginning or at the end of the string, respectively. Then, a tree for each snippet x_s^{behind}, x_s, and x_s^{ahead} is built as in a. Finally, a tree is built such that it corresponds to the concatenation of (i) a lookbehind node whose child is the tree obtained from x_s^{behind}, (ii) the tree obtained from x_s, and (iii) a lookahead node whose child is the tree obtained from x_s^{ahead}.
d) A tree is generated as in c and then modified as in b, by compacting subtrees of repeated leaf nodes. For lookbehind trees, subtrees are replaced by l{1,m}+, rather than l++, where l is the repeated label and m is the number of its occurrences in the subtree. This change is made to accommodate a limitation of common regular expression libraries which do not allow for ++ and *+ to occur within lookbehinds.

If the number of individuals generated from the training set $\mathcal{T}$ is greater than n_{pop}, exceeding individuals are removed at random; otherwise, missing individuals are generated at random with a Ramped half-and-hald method [16], each one with a tree depth chosen randomly within the interval 2–15. Whenever an individual is generated whose string transformation is not a valid regular expression, it is discarded and a new one is generated.

Once the initial population is built, it is evolved iteratively as follows. At each iteration (called *generation*), n_{pop} new individuals are generated: 80% by *crossover* of pairs of individuals of the current population, 10% by *mutation* of individuals of the current population, 10% generated randomly with a Ramped half-and-half method. Crossover is a genetic operator which takes two individuals and outputs two new individuals that are identical to the input individuals except for two randomly selected subtrees that are swapped. Mutation is a genetic operator which takes an individual and outputs a new individual identical to the input inividual except for a randomly selected subtree that is replaced by a new randomly generated subtree. The choice of an individual (or a pair of individuals) to undergo mutation (or crossover) is made with a *tournament selection*: 7 individuals are randomly picked in the current population and the one with the best fitness is selected. A new population is built from the resulting $2n_{\text{pop}}$ individuals,

$s_1 =$ height:␣174.5cm $s_2 =$ height:␣n.a.

$x_{s_1} = 174.5_8$

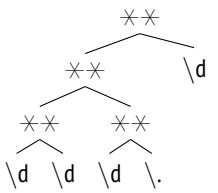
$r_a =$ \d\d\d\.\d

$r_b =$ \d++\.\d

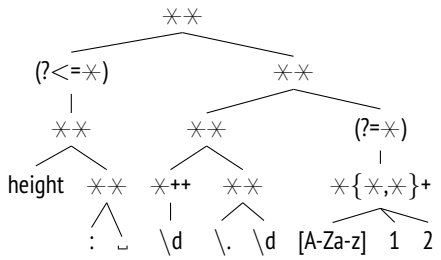
$r_c =$ (?<=height:␣)\d\d\d\.\d(?=[A-Za-z][A-Za-z])

$r_d =$ (?<=height:␣)\d++\.\d(?=[A-Za-z]{1,2}+)

(a) Individuals generated for x_s.



(b) Tree for r_a.



(c) Tree for r_d.

Fig. 1: Example of the population initialization from a training set of 2 examples: 4 individuals r_a, r_b, r_c, r_d are generated from the only desired extraction x_{s_1}. The trees corresponding to 2 of them are shown: note that, in r_d, height is a token label (see Section 4.1.1). The subscript of the individuals (a–d) corresponds to the specific points described in Section 4.1.3.

by retaining only the best n_{pop} of them.

The above procedure includes a *genotypic diversity enforcement* criterion which was not present in [15] (a very similar mechanism is used in [43]): whenever an individual r_1 is generated whose string transformation is the same as one of an existing individual r_2 (i.e., one in the current population or one previously generated in the current iteration), r_1 is discarded and a new one is generated.

The iterative procedure is repeated until one of the two following conditions is met: (i) a number of n_{gen} iterations have been performed, or (ii) the fitness tuple of the best individual has remained unchanged for n_{stop} consecutive iterations. The string transformation of the best individual of the population at the end of the last iteration is the outcome of the GP search.

4.1.4 Separate-and-conquer

A problem instance may include desired extractions which are structurally very different from each other. 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. While problem instance of this sort could be theoretically addressed by including the or operator | among the building blocks available to the GP search for building candidate solutions, in practice such a design choice is ineffective. As it turned out from our analyses, that we omit for brevity, inclusion of the or operator generally leads to poor solutions, probably because of the much increased size of the solution space along with the difficulty of figuring out when such operator is actually needed and at which exact point of a candidate solution.

To address this important practical problem we designed a *separate-and-conquer* search procedure (which we previously sketched in [20]) that does not require the or operator yet is able to realize automatically whether multiple patterns are required and, in that case, to actually generate such patterns with an appropriate trade-off between specificity and generality.

A separate-and-conquer search consists of an iterative procedure in which, at each iteration, a GP search is performed and the snippets correctly extracted by the set of regular expressions generated so far are removed from the training set for the next iteration. This general scheme [17] is useful to cope with scenarios in which several problem sub-instances that are not explicitly delimited could be identified, such as in various forms of rule inference [18], [19], [44]. In detail, initially the target regular expression $\hat{r}$ is set to the empty string, then the following sequence of steps is repeated:

1) Perform a GP search on $\mathcal{T}$ and obtain r.
2) If $\text{Prec}(r, \mathcal{T}) = 1$, then assign $\hat{r} := \hat{r}|r$ (i.e., concatenate $\hat{r}$, the regular expression *or* operator |, and r), otherwise terminate.
3) For each $(s, X_s^d, X_s^u) \in \mathcal{T}$, assign $X_s^d := X_s^d \setminus [\mathcal{X}_s]_{\hat{r}}$;
4) If $\bigcup_{(s,X_s^d,X_s^u)\in\mathcal{T}} X_s^d$ is empty, then terminate.

In other words, at each iteration we require the currently generated regular expression r to have perfect precision (step 2): i.e., r must extract only snippets which are indeed to be extracted, but it might miss some other snippets. Since $\hat{r}$ is built up with the or operator, it extracts every snippet which is extracted by at least one of its components: it follows that $\hat{r}$ will have perfect precision and a recall greater than each of its components. The constraint on perfect precision of step 2 is indeed the reason for which we chose to favor the precision among individuals of the same Pareto frontier (see Section 4.1.2): the most prominent objective is exactly to maximize $\text{Prec}(p, \mathcal{T})$. Subsequent iterations will target the snippets still missed by $\hat{r}$ (step 3).

The GP search at step 1 is performed with $n_{\text{stop}} \ll n_{\text{gen}}$, so as to leave "difficult" examples for subsequent iterations of the separate-and-conquer procedure (by allowing an *early termination* of the search) and to avoid to over-focus on a training set when no significant improvements appear to be achievable.

4.2 High level organization of GP searches

Since a GP execution is a stochastic procedure, we follow a common approach in evolutionary algorithms which consists in executing multiple independent searches on the same training set and then selecting one of the solutions according to a predefined criterion.

In detail, we proceed as follows.

1) We partition the set E of examples available for learning in two sets E_t and E_v.
2) We build the training set $\mathcal{T}$ of the GP search based on E_t (see below) and keep E_v not available to the search.
3) We execute $2n_{job}$ independent GP searches, all with the same $\mathcal{T}$ but each with a different random seed. We call

each such search a *job*. Execution of this step generates a pool of $2n_{\text{job}}$ solutions.

4) We compute the F-measure of each of the $2n_{\text{job}}$ solutions on the full set of learning examples $E = E_t \cup E_v$ and select the solution with best F-measure.

In other words, we use E_v as a *validation set* for assessing the generalization ability of a proposed solution on examples that were not available to the learning process —i.e., to prevent overfitting while promoting generalization.

The partitioning of E is made randomly so that the number of the snippets in E_t and E_v are roughly the same, i.e., $\sum_{(s,X_s)\in E_t} |X_s| \approx \sum_{(s,X_s)\in E_v} |X_s|$. The training set $\mathcal{T}$ for jobs is built simply: for each $(s, X_s) \in E_t$, a triplet $(s, X_s, \{s_0\} \ominus X_s)$ is inserted in $\mathcal{T}$ (i.e., $X_s^d := X_s$ and $X_s^u := \{s_0\} \ominus X_s$).

In order to broaden the spectrum of problem instances that can be addressed effectively, we do not execute all the $2n_{\text{job}}$ jobs in the same way. Instead, we execute n_{job} jobs according to separate-and-conquer, while each of the other n_{job} jobs consist of a *single* GP search where all the available generations (i.e., $n_{\text{stop}} = n_{\text{gen}}$) are devoted to learning a single pattern on the full training set $\mathcal{T}$.

5 Experimental evaluation

We carried out a thorough experimental evaluation for addressing the following questions: 1) How does our method perform on realistic problem instances, even w.r.t. manual authorship of regular expressions? 2) How do other relevant methods perform compared to ours? 3) What is the role of some of the key features of our proposal? We analyze each question in the following subsections.

We implemented the method here proposed as a Java application[4] in which jobs are executed in parallel. The implementation includes some significant optimizations aimed at speeding up executions: a full description can be found in our technical report [45]. We tuned the values for the parameters n_{job}, n_{pop}, n_{gen} and n_{stop} (the latter actually matters only in separate-and-conquer jobs) after exploratory experimentation and taking into account the abundant state of the art about GP. We set $n_{\text{job}} = 16$ (4 for the web version), $n_{\text{pop}} = 500$, $n_{\text{gen}} = 1000$ and $n_{\text{stop}} = 200$.

5.1 Extraction tasks and datasets

We considered 20 different extraction tasks defined by relevant combinations of 17 entity types to be extracted from 12 text corpora. We made available[5] part of the extraction tasks: we excluded those previously used in other works and those which cannot be included for privacy issues (e.g., those containing email addresses).

Table 1 (four leftmost columns) shows salient information about the 20 extraction tasks: number of examples $|E_0|$, their overall length (in thousands of characters) $\sum_{(s,X_s)\in E_0} \ell(s)$, and overall number of snippets $\sum_{(s,X_s)\in E_0} |X_s|$. The name of each extraction task is composed of the name of the corpus (see below) followed by the name of the entity type to be extracted. Entity names should be self-explanatory: Username corresponds to extracting only the username from Twitter citations (e.g., only MaleLabTs instead of @MaleLabTs); Email-ForTo corresponds to extracting email addresses appearing after the strings for: or to: (possibly capitalized). It seems fair to claim that these extraction tasks are quite challenging and representative of real world applications. Names ending with a $*$ suffix indicate extraction tasks with context (Section 3.2.1).

The text corpora are listed below. Some of them have been used in previous works about text extraction with the same (or similar) entity types to be extracted—all corpora but the last three ones have been used also in [15].

ReLIE-Web: portions of several web pages from the publicly available University of Michigan Web page collection. Used in [10].

ReLIE-Email: portions of the body of several emails from the publicly available Enron email collection. Used in [10], [13].

Cetinkaya-HTML: lines of the HTML source of 3 web pages. Used in [24].

Cetinkaya-Web: lines of plain text taken from 3 web pages after rendering. Used in [24].

Twitter: 50 000 Twitter messages collected using the Twitter Streaming API.

Log: 20 000 log entries collected from our lab gateway server running the vuurmuur firewall software.

Email-Headers: 101 headers obtained from several emails collected from personal mail boxes of our lab staff.

NoProfit-HTML: lines of the HTML source of the address book web page of a local nonprofit association.

Web-HTML: lines of the HTML source of several pages.

CongressBills: 600 US Congress bills from the THOMAS online database. In order to vary the format of the dates, we changed the format of the dates as to obtain 9 different formats—including 3 formats in which the month is expressed by name rather than by number.

BibTeX: 200 bibliographic references in the form of BibTeX elements obtained with Google Scholar.

Reference: 198 bibliographic references (the same of the BibTeX corpus with two removals) formatted according to the Springer LNCS format.

5.2 Proposed method effectiveness

We evaluated our method as follows. For each extraction task we built several problem instances $(E, E^\star)$ differing in the overall number of snippets $\sum_{(s,X_s)\in E} |X_s|$ available for learning. In each problem instance we partitioned the set of examples E_0 in a learning set E and a testing set $E^\star = E_0 \setminus E$. We experimented with the values $24, 50, 100$ for the number of snippets in E. We applied our method 5 times for each of those values, by randomly varying the composition of E and hence $E^\star$, and averaged the obtained figures of precision and recall over the 5 repetitions. Hence, we analyzed 300 problem instances—5 repetitions for each of the 60 different combinations of extraction task and number of snippets for learning.

Table 1 summarizes our main results. The table has 60 rows, one for each combination of extraction task and number of snippets for learning. Sixth and seventh columns

4. The source code is available on https://github.com/MaLeLabTs/RegexGenerator. A web-based version of the application is available on http://regex.inginf.units.it.

5. http://machinelearning.inginf.units.it/data-and-tools/annotated-strings-for-learning-text-extractors

contain the number of snippets for learning and the *learning ratio (LR)* defined as the ratio between the number of snippets for learning and the number of snippets in the full extraction task E_0. The remaining columns on the left illustrate performance indexes of the learned regular expression $\hat{r}$: F-measure on the learning data E and, most importantly, precision, recall, and F-Measure on the testing data $E^\star$. The two last columns provide indexes for assessing the computational effort: EC is the overall number of characters which have been evaluated by candidate regular expressions—e.g., a population of 100 individuals applied to a set E including strings totaling 1000 characters for 100 generations corresponds to EC $= 10^7$. Figures in the table are expressed in multiples of 10^{10}. TtL is the time required for solving a problem instance: we used a machine powered with a 6 core Intel Xeon E5-2440 ($2.40\,\mathrm{GHz}$) equipped with $32\,\mathrm{GB}$ of RAM.

The key outcome of this experimental campaign is that F-measure on testing data is very high in nearly all scenarios analyzed. This result is particularly relevant in itself and becomes even more relevant in light of the very low LR values of our experimental setting, which indicate that our method is indeed able to find solutions that generalize effectively. It can also be seen that, in many extraction tasks, F-measure is very high also when the learning information includes only 24 snippets. This suggests that the proposed method can be very effective even with *few* examples.

The only extraction task in which F-measure is definitely unsatisfactory—in the range 35.8–48.3%—is ReLIE-Email/Phone-Number. This task was executed with LR in the range 0.5–1.9%. We executed this task again with LR $\approx 80\%$, a value much closer to the values usually used in machine learning literature and obtained a much higher F-measure on the testing data $E^\star$: $\approx 85\%$. We believe that this result demonstrates the quality of our approach even for this task. We carefully analyzed the results for ReLIE-Email/Phone-Number and we believe that this task is unlikely to be solved effectively with a very low LR. In particular, it can be seen from Table 1 that the generated regular expressions exhibit a rather high recall (92.6–98.3%) and a low precision (37.1–22.7%) on $E^\star$—i.e., they tend to extract all the relevant snippets but also unrelevant portions of the strings in $E^\star$. We manually inspected the learning data E and verified that they are not adequately representative of the data that are *not* to be extracted: they did not contain substrings which look like, but are not, phone numbers.

Concerning the impact of the number of snippets available for learning, results of Table 1 generally confirm that the more information available for learning, the better the obtained F-measure. There are indeed a few anomalies to this trend which, we believe, are due to the very low LR values and the highly challenging nature of the extraction tasks.

With respect to the *computational effort* (i.e., EC and TtL), our experimental evaluation shows that the time needed to learn a regular expression for a problem instance is often in the order of a few tens of minutes. We also found, as expected, that TtL depends approximately linearly from EC, which itself strongly depends on the aggregate "size" of the learning information in terms of characters, i.e., $\sum_{(s,X_s)\in E} \ell(s)$. While the absolute value of TtL would seem to discourage the on-the-fly usage of our method, our experience with its web-based implementation suggests that TtLs do not hamper the practicality of our tool. Moreover, we believe that TtL should be assessed from a *relative* point of view: a user highly skilled in regular expression writing probably would not even use our tool, while a user moderately skilled or unskilled at all may solve problems that would otherwise be unable to solve—see also Section 5.3.

We found that tasks which may take advantage of modern regular expression constructs (lookarounds, possessive quantifiers) tend to require a longer execution time. We think this finding is motivated by the fact that our tool operates with a real-world regular expression engine (the one included in the Java platform): that engine cannot guarantee that the processing time of *every* regular expression grows linearly with the input string length, because the previously mentioned constructs cannot be implemented using automata; it follows that tasks in which the evolution tends to favor regular expressions with modern constructs, take much longer times to be solved.

A list of the generated regular expressions is available in our technical report [45].

5.3 Comparison with human operators

In order to assess the ability of our method to compete with human operators, we executed an experiment using a web application which we crafted ad hoc.

The web app presented concise instructions about the experiment ("write a regular expression for extracting text portions which follow a pattern specified by examples") and then asked the user to indicate the level of familiarity with regular expressions—one among novice, intermediate, and experienced. The web app then proposed a sequence of extraction tasks: for each task the web app showed a text on which the snippets to be extracted were highlighted; the user could write and modify a regular expression in an input field at will; the web app immediately highlighted the snippets actually extracted by the current expression along with the corresponding extraction mistakes (if any). The web app also showed the F-measure and the user was informed that a value of 100% meant a perfect score on the task. The user was not required to obtain a perfect F-measure before going to the next task—i.e., he could give up on a task. In the limit, he could also not write any regular expression for a task (*unanswered task*). The web app recorded, for each task and for each user, the authored regular expression and the overall time spent.

We included in the web app 9 of the extraction tasks presented in the Section 5.1. For each task, we chose exactly the E set we used while experimenting with our method for repetition 1 and $\sum_E |X_s| = 24$. We spread a link to the web app among CS graduate and undergraduate students of our University. Each user interacted with the web app autonomously and in an unconstrained enviroment—in particular, users were allowed to (and not explicitly instructed not to) refer to any knowledge base concerning regular expressions.

We gathered results from 73 users—60% novice, 20% intermediate, and 20% experienced. Several tasks were left

TABLE 1: Results and salient information about the extraction tasks. The overall length $\sum_{(s,X_s)\in E_0} \ell(s)$ of examples is expressed in thousands of characters. EC is expressed in 10^{10} evaluated characters; TtL is expressed in minutes.

						On E		On $E^\star$			
Extraction task E_0	$\lvert E_0\rvert$	$\sum_{E_0} \ell(s)$	$\sum_{E_0} \lvert X_s\rvert$	$\sum_{E} \lvert X_s\rvert$	LR	Fm	Prec	Rec	Fm	EC	TtL
ReLIE-Web/All-URL	3877	4240	502	24	5.0	99.2	90.0	91.9	90.9	2.6	15
				50	10.0	99.2	92.1	95.0	93.5	6.4	35
				100	19.9	98.9	94.8	96.5	95.6	13.7	71
ReLIE-Web/HTTP-URL	3877	4240	499	24	5.0	99.2	86.3	89.0	87.6	2.5	11
				50	10.0	99.0	91.0	93.3	92.2	5.8	32
				100	20.0	98.8	92.9	96.8	94.8	13.1	66
ReLIE-Email/Phone-Number	41 832	8805	5184	24	0.5	97.7	37.1	92.6	48.3	3.4	8
				50	1.0	99.0	29.9	96.6	43.3	6.0	16
				100	1.9	98.9	22.7	98.3	35.8	14.4	39
Cetinkaya-HTML/href	3425	154	214	24	11.7	100.0	98.7	99.2	98.9	2.5	12
				50	23.4	100.0	98.1	98.7	98.4	4.9	26
				100	46.7	99.8	98.4	99.1	98.8	9.0	59
Cetinkaya-HTML/href-Content*	3425	154	214	24	11.7	98.4	74.9	98.7	80.6	2.4	16
				50	23.4	98.5	85.1	98.8	88.2	4.8	29
				100	46.7	98.5	83.2	96.8	86.2	10.5	67
Cetinkaya-Web/All-URL	1234	39	168	24	14.9	99.2	99.4	98.8	99.1	1.7	3
				50	29.8	100.0	95.5	98.6	96.9	3.2	8
				100	59.5	99.5	98.8	98.8	98.8	5.2	16
Twitter/Hashtag+Citation	50 000	4344	56 994	24	0.1	100.0	98.8	100.0	99.4	1.2	3
				50	0.1	99.6	99.2	100.0	99.6	2.2	4
				100	0.2	99.8	99.0	100.0	99.5	4.6	7
Twitter/All-URL	50 000	4344	14 628	24	0.2	100.0	94.7	98.5	96.6	1.8	3
				50	0.3	100.0	96.2	98.3	97.2	3.4	8
				100	0.7	99.4	96.1	98.0	97.0	7.7	16
Twitter/Username*	50 000	4344	42 352	24	0.1	100.0	99.3	100.0	99.7	1.2	2
				50	0.1	100.0	99.2	100.0	99.6	2.2	2
				100	0.2	99.9	99.3	100.0	99.7	4.6	2
Log/IP	20 000	4126	75 958	24	0.1	100.0	99.8	100.0	99.9	1.3	2
				50	0.1	100.0	99.7	100.0	99.8	2.3	2
				100	0.2	100.0	99.8	100.0	99.9	4.6	3
Log/MAC	20 000	4126	38 812	24	0.1	100.0	100.0	100.0	100.0	2.0	2
				50	0.1	100.0	100.0	99.4	99.7	4.3	3
				100	0.3	100.0	100.0	99.4	99.7	8.3	6
Email-Headers/IP	101	261	848	24	2.9	97.5	86.7	87.9	86.9	5.9	18
				50	5.9	92.7	90.9	82.2	86.0	14.0	56
				100	11.8	94.5	95.2	84.9	89.6	28.5	89
Email-Headers/Email-ForTo*	101	261	331	24	7.6	78.5	70.7	52.5	59.3	17.9	131
				50	15.1	71.5	76.4	52.8	62.0	33.7	398
				100	30.2	79.8	90.4	66.6	76.4	65.5	429
NoProfit-HTML/Email	25 590	860	1094	24	2.3	100.0	83.2	100.0	85.5	0.9	2
				50	4.6	100.0	100.0	100.0	100.0	1.9	3
				100	9.1	100.0	100.0	100.0	100.0	3.7	7
Web-HTML/Heading	49 026	4541	1083	24	2.3	99.2	93.1	89.4	91.2	7.6	30
				50	4.6	96.2	93.3	90.2	91.7	15.3	83
				100	9.2	99.2	98.2	96.2	97.2	29.7	256
Web-HTML/Heading-Content*	49 026	4541	1083	24	2.3	93.6	95.5	80.1	86.6	6.6	76
				50	4.6	95.9	99.1	85.8	91.8	13.6	168
				100	9.2	98.9	99.4	96.1	97.7	28.0	379
CongressBill/Date	600	16 511	3085	24	0.8	64.5	57.1	52.3	50.0	2.1	30
				50	1.6	72.1	55.4	81.3	64.1	6.9	584
				100	3.2	76.1	62.7	81.4	69.7	11.3	513
BibTeX/Title	200	54	200	24	12.5	89.6	79.1	65.1	70.7	5.1	43
				50	25.0	90.3	82.6	74.3	78.0	11.1	141
				100	50.0	82.0	84.8	63.4	72.1	21.5	218
BibTeX/Author	200	54	589	24	4.2	92.9	90.5	78.1	83.1	2.0	8
				50	8.5	93.9	89.9	86.1	87.7	4.1	20
				100	17.0	90.7	91.9	81.6	86.2	7.5	34
References/First-Author*	198	30	198	24	12.6	99.0	99.7	96.0	97.8	2.8	12
				50	25.3	96.3	99.6	93.6	96.5	5.4	26
				100	50.5	100.0	100.0	100.0	100.0	12.4	56

TABLE 2: F-measure for $\sum_E |X_s| = 24$ obtained by human operators (novice (SN), intermediate (SI), and experienced (SE)) and our approach (O).

Extraction task	SN	SI	SE	O
ReLIE-Web/All-URL	74.7	90.2	80.6	95.5
ReLIE-Web/HTTP-URL	77.3	83.0	76.6	82.3
ReLIE-Email/Phone-Number	70.2	84.7	91.0	34.6
Cetinkaya-HTML/href	91.6	98.8	98.8	100.0
Cetinkaya-Web/All-URL	95.2	98.3	98.6	99.0
Log/IP	91.0	100.0	100.0	100.0
Log/MAC	87.6	91.7	100.0	100.0
Web-HTML/Heading	82.3	90.9	95.6	90.0
BibTeX/Author*	64.6	50.1	81.4	90.3

unanswered: 42% for novice, 40% for intermediate, and 12% for experienced. The average time for solving the answered tasks was $16.1\,\mathrm{min}$, $4.8\,\mathrm{min}$, and $4.7\,\mathrm{min}$, respectively. As a comparison, our method on the very same data required TtL $= 10.4\,\mathrm{min}$ on the average.

The key finding is in Table 2, which shows the F-measure on $E^\star$ for each task. It can be seen that the F-measure obtained by our method is almost always greater than or equal to the one obtained by human users (on the average). The only exceptions are: the ReLIE-Email/Phone-Number task (whose peculiarity has been analyzed in Section 5.2); the Web-HTML/Heading task, in which our method improves over novice users and is only sligthly worse than intermediate users. We believe this result is remarkable and highly encouraging. Indeed, we are not aware of any proposal for automatic generation of regular expressions in which human operators were used as a baseline. A full description of the results can be found in the companion report [45].

5.4 Comparison with other methods

The previous section considered a baseline in terms of human operators. In this section we consider a baseline in terms of other approaches for learning text extractors from examples: *Smart State Labeling DFA Learning (SSL-DFA)* [46], *FlashExtract* [33], and *GP-Regex* [15]. These methods are representative of the state of the art for learning syntactical patterns (see also Section 2), but differ in the actual nature of the learned artifact: SSL-DFA produces Deterministic Finite Automata (DFA), FlashExtract produces extraction programs expressed in a specific language, and GP-Regex produces regular expressions. Results obtained with SSL-DFA were significantly worse than those of the other methods, hence we chose to not describe them in this paper: full details can be found in our technical report [45]. We remark that GP-Regex was compared against the approaches of [10], [13] on two datasets used by the latter and exhibited better accuracy, even with a learning set smaller by more than one order of magnitude [15]; and, that the authors of [10], [13] showed that their approaches exhibited performance similar to *Conditional Random Fields* (CRFs). We did confirm the superiority of our approach w.r.t. CRFs, in terms of both F-measure and TtL, with a brief experimental comparison (we omit here the results due to space constraints).

FlashExtract is a powerful and sophisticated framework for extracting multiple different fields automatically in semi-structured documents [33]. It consists of an inductive synthesis algorithm for synthesizing data extraction programs from few examples, in which programs are expressed in any underlying domain-specific language supporting a predefined algebra of few core operators. The cited work presents also a language designed to operate on text which perfectly fits the extraction problem considered in this paper. The findings of [33] resulted in a tool included in the Windows Powershell as the ConvertFrom-String command: we used this tool to perform the experiments. The current FlashExtract implementation does not allow reusing a program induced by a given set of examples. Thus, in our experimentation the two phases of learning and testing were not separated: we invoked the tool by specifying as input the examples in E and the strings in $E^\star$; we obtained as output a set of substrings extracted from $E^\star$ based on the description in E (which we had to recast in the syntax required by the tool). In many cases the tool crashed, thereby preventing the extraction to actually complete. We highlighted these cases in the results.

TABLE 3: F-measure for $\sum_E |X_s| = 100$ with FlashExtract (F), GP-Regex (G), and our approach (O).

Extraction task	F	G	O
ReLIE-Web/All-URL	21.5	93.0	95.6
ReLIE-Email/Phone-Number	–	90.2	35.8
Cetinkaya-HTML/href	32.3	89.6	98.8
Cetinkaya-Web/All-URL	61.8	94.9	98.8
Twitter/Hashtag+Citation	–	100.0	99.6
Web-HTML/Heading-Content*	–	10.2	97.7
CongressBill/Date	–	38.0	70.7

GP-Regex is the method we proposed in [15] and the base for the research here presented. The numerous differences between our method and GP-Regex were listed in the introduction. We emphasize again that in GP-Regex each example consists of a string and at most one single snippet to be extracted from that string. In order to build learning examples suitable for GP-Regex, we considered for each (s, X_s), only the leftmost snippet in X_s, if any.

We selected 7 extraction tasks including tasks with context and tasks in which snippets exhibit widely differing formats. We exercised all methods with the same experimental settings described in Section 5.1, thereby obtaining 105 problem instances—5 repetitions for each of the 21 different combinations of extraction task and number of snippets for learning.

Table 3 shows the results in terms of F-measure for $\sum_E |X_s| = 100$—results for other values for $\sum_E |X_s|$ are omitted due to space constraints. The foremost outcome of this comparison is that our method clearly outperforms all the other methods (except for ReLIE-Email/Phone-Number, discussed below). The performance gap with FlashExtract is substantial—at the expense of a much longer TtL, though. We are not able to provide any principled interpretation for this result. We may only speculate that our approach is perhaps more suitable for coping with loosely structured or unstructured datasets than FlashExtract. We also noticed that, for many problem instances, the ConvertFrom-String tool crashed, thereby preventing the extraction to actually complete. For the extraction tasks for which at least one on 5 repetition completed without errors, Table 3 shows the F-measure averaged across the completed executions.

TABLE 4: F-measure for $\sum_E |X_s| = 100$ with our fitness (O) and with the F-measure based fitness (F).

Extraction task	O	F	ΔFm
ReLIE-Web/All-URL	95.6	11.7	83.9
ReLIE-Web/HTTP-URL	94.8	14.8	80.0
Cetinkaya-HTML/href	98.8	98.6	0.2
BibTeX/Title*	72.1	3.3	68.8
BibTeX/Author*	86.2	24.9	61.3
References/First-Author*	100.0	NaN	NaN

TABLE 5: F-measure with $\sum_E |X_s| = 100$ with and without initialization.

Extraction task	w/	w/o	ΔFm
ReLIE-Web/All-URL	95.6	73.6	22.0
ReLIE-Web/HTTP-URL	94.8	82.6	12.2
Cetinkaya-HTML/href	98.8	48.8	50.0
BibTeX/Title*	72.1	65.1	7.0
BibTeX/Author*	86.2	67.2	19.0
References/First-Author*	100.0	78.7	21.3

In the other cases, we were not able to obtain any extraction program, neither splitting the testing set in small chunks: those cases are denoted with an en dash (−) in the table.

Concerning GP-Regex, we should isolate two groups of extraction tasks: (i) those that requires either a context (Web-HTML/Heading-Content*) or the ability to learn widely differing patterns (CongressBill/Date), (ii) all the other tasks. The key observation is that our current proposal improves over GP-Regex in all cases (except for ReLIE-Email/Phone-Number), the improvement being substantial in case i. Indeed, our current proposal makes it possible to handle both Web-HTML/Heading-Content* and Congress-Bill/Date with good accuracy, while GP-Regex does not. It is also interesting to observe that in case ii GP-Regex provides much better accuracy than FlashExtract, while in case i GP-Regex is either comparable to those methods or worse.

Finally, concerning the ReLIE-Email/Phone-Number extraction task, we observe that this is the same task with a sort of anomalous behavior already discussed in the previous section. In particular, we remark that when executing our method on this task with LR $\approx 80\%$ we obtained $\approx 85\%$ F-measure on the testing data. We could not execute FlashExtract in those conditions because it always crashed: the only result that we could obtain is in Table 3, where F-measure (with very few examples available for learning) is 69%. The reason why GP-Regex happens to deliver better accuracy on this task is because it tends to overfit the snippets to be extracted more than the method here presented. As discussed in the previous section, processing this task with a very small LR value incurs in a poor representativeness of the text that is not to be extracted; as it turns out, thus, the slightly overfitting behavior exhibited by GP-Regex in this case turns out to be a pro.

5.5 Assessment of specific contributions

In order to gain further insights into our proposal, we executed a further suite of experiments on a subset of the extraction tasks aimed at assessing the effect of: (i) choice of the fitness, (ii) initialization of the population from E, and (iii) separate-and-conquer jobs.

5.5.1 Fitness

We built a variant of our method in which the fitness tuple of a solution consists in the F-measure on the examples in the training set and the length of the corresponding regular expression: $f(r) := (\text{Fm}(r, \mathcal{T}), \ell(r))$. In other words, we replace snippet-level precision and character-level accuracy (see Section 4.1.2 for the exact definition) by snippet-level F-measure, i.e., by the main performance index desired by the solution.

Table 4 presents the results. The rightmost column shows the improvement ΔFm obtained by our proposal w.r.t. the method with the fitness modified as above. It can be seen that the modified method leads to a much worse F-measure, despite F-measure being exactly the index optimized by that method: our proposal leads to an improvement, on the average, around $\Delta\text{Fm} \approx 60\%$ ($\sum_E |X_s| = 100 \in \{24, 50, 100\}$). In other words, driving the evolutionary search by the key index of interest is not the optimal fitness choice. This finding corroborates some arguments made in [15] and augments them with an experimental evaluation.

It is worth to note that for the References/First-Author* task, the modified method is simply unable to produce a solution which can correctly extract at least one snippet. Our explanation is that the solving regular expression for that task is rather complex, since it includes multiple lookaround operators: light modifications to a regular expression which includes operators of this kind may result in very different extraction behaviors. In such a case, a fitness based on full snippets rather than individual characters does not acknowledge for small improvements and is not hence able to drive the evolution—in other words, it imposes an excessive evolutionary pressure.

5.5.2 Initialization

We built a variant of our method in which the initial population is totally built at random, instead of being partially generated using the snippets of the training set.

Table 5 shows the comparison results, which clearly indicate that the unmodified version is much more effective ($\Delta\text{Fm} \approx 25\%$, on the average for $\sum_E |X_s| \in \{24, 50, 100\}$). The rationale of the population initialization from the examples was to start the evolutionary search from a "good" point in the solution space. For this reason we inserted in the initial population individuals which fitted the snippets to be extracted while at the same time generalizing beyond them, e.g., we insert the regular expression `\d++-\d++-\d++` from the snippet `07-02-2011` (see Section 4.1.3).

5.5.3 Separate-and-conquer

We built a variant of our method in which all the $2n_{\text{job}}$ jobs are executed without the separate-and-conquer strategy, i.e., all jobs consist of a single GP search for which $n_{\text{stop}} = n_{\text{gen}}$.

Table 6 shows the comparison results. For this comparison, we considered also an extraction task (Congress-Bill/Date) in which the snippets to be extracted exhibit widely differing formats. As expected, the unmodified method clearly outperforms the modified one on Congress-Bill/Date ($\Delta\text{Fm} \approx 30\%$ for $\sum_E |X_s| \in \{24, 50, 100\}$). On the other hand, it can be seen that some not negligible improvement can be obtained also for other tasks, namely

TABLE 6: F-measure with $\sum_E |X_s| = 100$ with and without separate-and-conquer.

Extraction task	w/	w/o	ΔFm
ReLIE-Web/All-URL	95.6	89.5	6.1
ReLIE-Web/HTTP-URL	94.8	87.6	7.2
Cetinkaya-HTML/href	98.8	95.3	3.5
CongressBill/Date	69.7	32.7	37.0
BibTeX/Title*	72.1	62.0	10.1
BibTeX/Author*	86.2	71.5	14.7
References/First-Author*	100.0	97.3	2.7

BibTeX/Title* and BibTeX/Author*. We think that the motivation is in that those tasks are more difficult and hence the possibility, enabled by the separate-and-conquer strategy, to split a problem in smaller subproblems may allow the method to better cope with such difficulty.

6 Conclusions

We have described a system for synthesizing a regular expression automatically, based solely on examples of the desired behavior. The regular expression is meant to be used for extraction problems of practical complexity, from text streams that are either loosely structured or fully unstructured. As such, our approach is able to handle potentially large alphabets effectively, thereby overcoming one of the principal limitations of much existing work in this area, and has been designed to address such practical needs as context-dependent extractions, widely different formats, and potentially large and unsegmented input streams.

We have analyzed our proposal experimentally in depth, by applying it on 20 challenging extraction tasks of realistic size and complexity, with a very small portion of the dataset available for learning. The results have been very good and compared very favorably with significant baseline methods. Most importantly, the results are highly competitive also with respect to a pool of more than 70 human operators.

We made publicly available the source code of our system (https://github.com/MaLeLabTs/RegexGenerator) and deployed an implementation as a web app (http://regex.inginf.units.it).

While our work may certainly be improved and enriched in several ways—faster learning, interactive learning procedures capable of starting with a very small number of snippets, even better accuracy, just to name a few—we do believe that our work may constitute a useful solution to a practically relevant and highly challenging problem.

References

[1] A. Brāzma, "Efficient identification of regular expressions from representative examples," in *Proceedings of the Sixth Annual Conference on Computational Learning Theory*, ser. COLT '93. New York, NY, USA: ACM, 1993, pp. 236–242. [Online]. Available: http://doi.acm.org/10.1145/168304.168340

[2] 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*, Jun 1994, pp. 396–400 vol.1.

[3] B. Svingen, "Learning regular languages using genetic programming," *Proc. 3-rd Genetic Programming Conference*, pp. 374–376, 1998.

[4] F. Denis, "Learning regular languages from simple positive examples," *Machine Learning*, vol. 44, no. 1-2, pp. 37–66, 2001. [Online]. Available: http://dx.doi.org/10.1023/A%3A1010826628977

[5] H. Fernau, "Algorithms for learning regular expressions from positive data," *Information and Computation*, vol. 207, no. 4, pp. 521 – 541, 2009.

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

[7] K. J. Lang, B. A. Pearlmutter, and R. A. Price, "Results of the abbadingo one DFA learning competition and a new evidence-driven state merging algorithm," in *Grammatical Inference*. Springer, 1998, p. 1–12. [Online]. Available: http://link.springer.com/chapter/10.1007/BFb0054059

[8] O. Cicchello and S. C. Kremer, "Inducing grammars from sparse data sets: a survey of algorithms and results," *The Journal of Machine Learning Research*, vol. 4, pp. 603–632, 2003.

[9] J. Bongard and H. Lipson, "Active coevolutionary learning of deterministic finite automata," *The Journal of Machine Learning Research*, vol. 6, p. 1651–1678, 2005. [Online]. Available: http://machinelearning.wustl.edu/mlpapers/paper_files/BongardL05.pdf

[10] Y. Li, R. Krishnamurthy, S. Raghavan, S. Vaithyanathan, and H. Jagadish, "Regular expression learning for information extraction," in *Proceedings of the Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 2008, pp. 21–30.

[11] 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*, ser. AND '10. New York, NY, USA: ACM, 2010, pp. 43–50. [Online]. Available: http://doi.acm.org/10.1145/1871840.1871848

[12] K. Murthy, D. P., and P. Deshpande, "Improving recall of regular expressions for information extraction," in *Web Information Systems Engineering - WISE 2012*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, vol. 7651, pp. 455–467. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-35063-4_33

[13] F. Brauer, R. Rieger, A. Mocan, and W. M. 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*. ACM, 2011, pp. 1285–1294.

[14] A. Bartoli, G. Davanzo, A. De Lorenzo, M. Mauri, E. Medvet, and E. Sorio, "Automatic generation of regular expressions from examples with genetic programming," in *Proceedings of the 14th Annual Conference Companion on Genetic and Evolutionary Computation*, ser. GECCO '12. New York, NY, USA: ACM, 2012, pp. 1477–1478. [Online]. Available: http://doi.acm.org/10.1145/2330784.2331000

[15] A. Bartoli, G. Davanzo, A. De Lorenzo, E. Medvet, and E. Sorio, "Automatic synthesis of regular expressions from examples," *Computer*, vol. 47, no. 12, pp. 72–80, Dec 2014.

[16] J. R. Koza, "Genetic Programming: On the Programming of Computers by Means of Natural Selection (Complex Adaptive Systems)," 1992.

[17] J. Fürnkranz, "Separate-and-conquer rule learning," *Artificial Intelligence Review*, vol. 13, no. 1, pp. 3–54, 1999.

[18] G. L. Pappa and A. A. Freitas, "Evolving rule induction algorithms with multi-objective grammar-based genetic programming," *Knowledge and information systems*, vol. 19, no. 3, pp. 283–309, 2009.

[19] R. C. Barros, M. P. Basgalupp, A. C. de Carvalho, and A. A. Freitas, "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, pp. 1237–1244.

[20] A. Bartoli, A. De Lorenzo, E. Medvet, and F. Tarlao, "Learning text patterns using separate-and-conquer genetic programming," in *18th European Conference on Genetic Programming*. Springer Verlag, 2015.

[21] F. Ciravegna *et al.*, "Adaptive information extraction from text by rule induction and generalisation," in *International Joint Conference on Artificial Intelligence*, vol. 17, no. 1. LAWRENCE ERLBAUM ASSOCIATES LTD, 2001, pp. 1251–1256.

[22] T. Wu and W. M. Pottenger, "A semi-supervised active learning algorithm for information extraction from textual data," *Journal of the American Society for Information Science and Technology*, vol. 56, no. 3, pp. 258–271, 2005.

[23] B. Rozenfeld and R. Feldman, "Self-supervised relation extraction from the web," *Knowledge and Information Systems*, vol. 17, no. 1, pp. 17–33, 2008.

[24] A. Cetinkaya, "Regular expression generation through grammatical evolution," in *Proceedings of the 2007 GECCO conference companion on Genetic and evolutionary computation*. ACM, 2007, pp. 2643–2646.

[25] Y. Xie, F. Yu, K. Achan, R. Panigrahy, G. Hulten, and I. Osipkov, "Spamming botnets: signatures and characteristics," in *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 4. ACM, 2008, pp. 171–182.

[26] P. Prasse, C. Sawade, N. Landwehr, and T. Scheffer, "Learning to identify regular expressions that describe email campaigns," in *Proceedings of the International Conference on Machine Learning*, 2012, arXiv preprint arXiv:1206.4637.

[27] D. D. A. Bui and Q. Zeng-Treitler, "Learning regular expressions for clinical text classification," *Journal of the American Medical Informatics Association*, vol. 21, no. 5, pp. 850–857, 2014.

[28] D. F. Barrero, M. D. R-Moreno, and D. Camacho, "Adapting searchy to extract data using evolved wrappers," *Expert Systems with Applications*, vol. 39, no. 3, pp. 3061–3070, Feb. 2012. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0957417411012991

[29] D. C. Smith, A. Cypher, and L. Tesler, "Programming by example: Novice programming comes of age," *Commun. ACM*, vol. 43, no. 3, pp. 75–81, Mar. 2000. [Online]. Available: http://doi.acm.org/10.1145/330534.330544

[30] R. A. Cochran, L. D'Antoni, B. Livshits, D. Molnar, and M. Veanes, "Program boosting: Program synthesis via crowd-sourcing," in *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '15. New York, NY, USA: ACM, 2015, pp. 677–688. [Online]. Available: http://doi.acm.org/10.1145/2676726.2676973

[31] P. Norvig, "xkcd 1313: Regex golf," http://nbviewer.ipython.org/url/norvig.com/ipython/xkcd1313.ipynb, Jan. 2014.

[32] A. Bartoli, A. De Lorenzo, E. Medvet, and F. Tarlao, "Playing regex golf with genetic programming," in *Proceedings of the 2014 Conference on Genetic and Evolutionary Computation*, ser. GECCO '14. New York, NY, USA: ACM, 2014, pp. 1063–1070. [Online]. Available: http://doi.acm.org/10.1145/2576768.2598333

[33] V. Le and S. Gulwani, "Flashextract: A framework for data extraction by examples," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2014, p. 55.

[34] J. Friedl, *Mastering regular expressions*. O'Reilly Media, Inc., 2006.

[35] S. Cronen-Townsend, Y. Zhou, and W. B. Croft, "Predicting query performance," in *Proceedings of the 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, ser. SIGIR '02. New York, NY, USA: ACM, 2002, pp. 299–306. [Online]. Available: http://doi.acm.org/10.1145/564376.564429

[36] C. Hauff, D. Hiemstra, and F. de Jong, "A survey of pre-retrieval query performance predictors," in *Proceedings of the 17th ACM Conference on Information and Knowledge Management*, ser. CIKM '08. New York, NY, USA: ACM, 2008, pp. 1419–1420. [Online]. Available: http://doi.acm.org/10.1145/1458082.1458311

[37] T. K. Ho and M. Basu, "Complexity measures of supervised classification problems," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 24, no. 3, pp. 289–300, Mar 2002.

[38] J.-R. Cano, "Analysis of data complexity measures for classification," *Expert Systems with Applications*, vol. 40, no. 12, pp. 4820 – 4831, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0957417413001413

[39] M. Graff, R. Poli, and J. J. Flores, "Models of performance of evolutionary program induction algorithms based on indicators of problem difficulty," *Evolutionary computation*, vol. 21, no. 4, pp. 533–560, 2013.

[40] K. Smith-Miles and L. Lopes, "Measuring instance difficulty for combinatorial optimization problems," *Computers & Operations Research*, vol. 39, no. 5, pp. 875–889, 2012.

[41] K. Smith-Miles, D. Baatar, B. Wreford, and R. Lewis, "Towards objective measures of algorithm performance across instance space," *Computers & Operations Research*, vol. 45, pp. 12–24, 2014.

[42] E. D. De Jong and J. B. Pollack, "Multi-objective methods for tree size control," *Genetic Programming and Evolvable Machines*, vol. 4, no. 3, pp. 211–233, 2003.

[43] W. Langdon and M. Harman, "Optimizing existing software with genetic programming," *Evolutionary Computation, IEEE Transactions on*, vol. 19, no. 1, pp. 118–135, Feb 2015.

[44] E. Medvet, A. Bartoli, B. Carminati, and E. Ferrari, "Evolutionary inference of attribute-based access control policies," in *Evolutionary Multi-Criterion Optimization*. Springer, 2015, pp. 351–365.

[45] A. Bartoli, A. De Lorenzo, E. Medvet, and F. Tarlao, "Inference of Regular Expressions for Text Extraction from Examples: supplementary material," DIA, University of Trieste, Italy, Tech. Rep., 2015. [Online]. Available: http://machinelearning.inginf.units.it/data-and-tools/inference-of-regular-expressions-for-text-extraction-from-examples

[46] S. M. Lucas and T. J. Reynolds, "Learning deterministic finite automata with a smart state labeling evolutionary algorithm," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 27, no. 7, pp. 1063–1074, 2005.

Alberto Bartoli received the degree in Electrical Engineering in 1989 cum laude and the PhD degree in Computer Engineering in 1993, both from the University of Pisa, Italy. Since 1998 he is an Associate Professor at the Department of Engineering and Architecture of University of Trieste, Italy, where he is the Director of the Machine Learning Lab. His research interests include machine learning applications, evolutionary computing, and security.

Andrea De Lorenzo received the diploma degree in Computer Engineering from the University of Trieste, Italy in 2006 and the MS degree in Computer Engineering in 2010. He received the PhD degree in Computer Engineering in 2014, endorsed by the University of Trieste. His research interests include evolutionary computing, computer vision, and machine learning applications.

Eric Medvet received the degree in Electronic Engineering in 2004 and the PhD degree in Computer Engineering in 2008, both from the University of Trieste, Italy. He is currently an Assistant Professor in Computer Engineering at the Department of Engineering and Architecture of University of Trieste, Italy, where he holds the position of Dean in Computer Engineering. His research interests include web and mobile security, genetic programming, and machine learning applications.

Fabiano Tarlao received the degree in Electronic Engineering from the University of Trieste, Italy in 2010. He is currently working toward the PhD degree from the Department of Engineering and Architecture at University of Trieste, Italy. His research interests are in the areas of web security, genetic programming, and machine learning applications.