Active Learning Approaches for Learning Regular Expressions with Genetic Programming

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

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

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

Fabiano Tarlao
DIA - University of Trieste
Italy
fabiano.tarlao@phd.units.it

ABSTRACT
We consider the long-standing problem of the automatic generation of regular expressions for text extraction, based solely on examples of the desired behavior. We investigate several active learning approaches in which the user annotates only one desired extraction and then merely answers extraction queries generated by the system.

The resulting framework is attractive because it is the system, not the user, which digs out the data in search of the samples most suitable to the specific learning task. We tailor our proposals to a state-of-the-art learner based on Genetic Programming and we assess them experimentally on a number of challenging tasks of realistic complexity. The results indicate that active learning is indeed a viable framework in this application domain and may thus significantly decrease the amount of costly annotation effort required.

CCS Concepts
• Information systems → Users and interactive retrieval;
• Mathematics of computing → Evolutionary algorithms;
• Computing methodologies → Learning paradigms;

Keywords
Information Extraction; Entity Extraction; Programming by Examples; Machine Learning

1. INTRODUCTION
A large class of entity extraction tasks from unstructured data 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. A long-standing problem in this area consists in the automatic generation of a regular expression suitable for a specific task based solely on examples of the desired behavior.

A wealth of research efforts in this area considered classification problems either in formal languages \[9, 15, 36, 14, 17, 20\] or in the realm of deterministic finite automata (DFA) \[22, 12, 7, 26\]. Those results considered scenarios that do not fit practical text processing applications, which have to cope with much longer sequences of symbols drawn from a much larger alphabet. Text extraction problems of non trivial size and complexity were first considered in a procedure that automatically optimized an initial regular expression to be provided by the user based on examples of the desired functioning \[24\]. Later proposals still required an initial regular expression but were more robust toward initial expressions of modest accuracy and noisy datasets \[3, 29\]. The need of an initial solution was later removed in several proposal \[11, 8, 4\]. A more recent proposal based on Genetic Programming advanced significantly over earlier approaches and is capable of addressing text extraction tasks of practical complexity effectively, with a few tens of examples of the desired behavior \[5, 6\].

In this work, we investigate the feasibility of an active learning approach for relieving the user from the need of examining the full input text (i.e., the dataset) in search of all the desired extractions to be annotated for learning \[1, 33, 35, 10, 25\]. We develop and evaluate experimentally a framework in which the user initially marks only one snippet of the input text as desired extraction. A learner based on Genetic Programming then constructs a solution, digs into the (possibly very long) input text, selects the most appropriate snippet to be used for improving the current model and presents it to the user as an extraction query. The user merely answers the query by specifying whether the selected snippet has to be extracted or not extracted and the process continues iteratively, improving the solution at each query.
I was born in 1979 and he was born in 1974.

\[ s_q \] I was born in 1979 and he was born in 1974.

\[ M, U \] I was born in 1979 and he was born in 1974.

Figure 1: Oracle annotation example: desired (undesired) extractions are in dark (light) gray; the query is boxed.

The resulting framework is highly attractive and may greatly broaden the potential scope of automatic regex generation from examples. On the other hand, actually implementing this framework is challenging because the scenario presents significant differences from successful applications of active learning.

Active learning approaches usually consider datasets where each item is an input instance and thus a candidate query. This property is shared also by approaches based on Genetic Programming [13, 30, 19]. Our case is different because the dataset is a single, possibly long, input text without any native segmentation in smaller units. Depending on the application, it may consist of one very long line or several lines with possibly variable length; furthermore, more than one desired extractions may occur within a single line (e.g., IP addresses in network logs) or a single desired extraction may span across several lines (e.g., HTML elements including their content). Assuming that the text is natively segmented in lines or in sentences (as in, e.g., [16]) would severely restrict the scope of possible applications of the system. Moreover, the size of the query to be presented to the user should be chosen carefully. Presenting a large snippet (e.g., one or more entire lines) to the user for annotation may nullify the objective of minimizing user annotation effort. On the other extreme, repeatedly asking the user to annotate very short snippets may not be effective.

In other words, not only we have the problem of choosing the next query among candidate queries, we also have the problem of constructing candidate queries out of the available input text. In this respect, it is useful to remark that the number of possible queries in (i.e., the number of snippets of) an input text grows quadratically with the text size and becomes huge very quickly—e.g., even if we assume that the learner cannot generate queries ex novo and can only query a snippet of the input text, if the latter size is just 10\(^5\) characters then there are \(\approx 10^{10}\) candidate queries. Furthermore, active learning usually targets scenarios with hundreds of queries (e.g., [25, 10, 25]) whereas we must be able to improve over a random query chooser and provide solutions of good quality even with a few tens of examples, similarly to [19].

Our contribution consists in: (a) a model for the external oracle that may participate in the construction of queries, which improves the quality of annotation information while at the same time maintaining a behavior very intuitive to unskilled users; (b) a technique for constructing queries suitable to regex-based entity extraction from unstructured text, which does not assume any internal segmentation of input text; (c) an implementation of several active learning approaches taking into account the need of constructing candidate queries; (d) a novel variant for the learner in which the number of generations executed between consecutive queries may vary dynamically depending on the quality of the current solution; and, (e) an experimental analysis on a number of challenging datasets of several active learning approaches, which target different accuracy/annotation effort regions of the design space.

2. OUR APPROACH

The problem consists in generating a regular expression automatically based on examples of the desired extraction behavior on a text \(t\). Such examples are annotations: snippets of \(t\) that are to be extracted (matches) or snippets of \(t\) that are not to be extracted (unmatches).

We propose an approach based on active learning, as follows. Initially an external oracle, i.e., the user, annotates an extremely small portion of \(t\)—we experimented with only one match. The learner consists of three components: the solver, which generates a regular expression suited to the matches and unmatches annotated by the oracle so far; the query trigger, which determines when a query has to be proposed to the oracle; and the query builder, which constructs candidate queries and determines which query should be proposed to the oracle.

Each query consists of a snippet of \(t\), denoted \(s_q\), to be annotated by the oracle. We propose the following behavior for the oracle: the oracle’s answer is a pair \(M, U\), where \(M\) is the (possibly empty) set of all matches which overlap \(s_q\) and \(U\) is the (possibly empty) set of maximal subsnippets of \(s_q\) which are unmatches—Figure 1 shows an example of annotation.

In other words, we propose an oracle that may modify the received query slightly and then answer the modified query. With most active learning approaches the user is required to provide the class of queried data and is not allowed to modify those data. The proposed behavior for the oracle is very practical and is easily implemented with a GUI, though. When the queried snippet consists exactly of a desired extraction or does not contain any desired extraction, one single click suffices to answer the query. Otherwise, when the query partly overlaps a match, the user is expected to expand the query on either or both sides—an action which is more intuitive to unskilled users, nevertheless results in answers which are more informative to the learner.

The solver is based on the proposal in [6], whose code is publicly available\(^1\). The proposal is based on Genetic Programming [21]: a population of regular expressions, represented by abstract syntax trees, is iteratively evolved by applying the genetic operators across many iterations (generations). A multiobjective optimization algorithm drives evolution of regular expressions according to their length (to be minimized) and their extraction performance computed on the matches and unmatches (to be maximized). We refer the reader to the cited paper for full details.

We considered two variants for the query trigger. The \(\texttt{Const}\) variant has been used in other active learning proposals for Genetic Programming [13, 30, 19] and generates a new query whenever a predefined number of generations of the solver has been executed. \(W\) experimented with 30 and with 200

\(^1\)https://github.com/MaLeLabTs/RegexGenerator
generations. The *Solved* variant is an optimization that we explore in this work. This variant triggers the query builder when the best regular expression in the population, as assessed on the current set of matches and unmatches, has remained unchanged for a predefined number of generations of the solver—i.e., a new query is triggered when no further progress seems to be achievable with the available annotations. We experimented with 200 generations, i.e., one of the values selected for the *Const* variant, in order to assess the accuracy/speed trade-off of the two variants.

The query builder constructs candidate queries based on the notion of *disagreement*: given a set *C* of regular expressions (the committee), we define as disagreement of *C* on a character *c* of the input text *t* the quantity $d_C(c) = 1 - 2 \text{abs} \left( \frac{1}{2} - \frac{|C_c|}{|C|} \right)$, where $C_c \subseteq C$ is the subset of regular expressions which extract *c*—$d_C(c) = 1$ if half of the committee extracts *c* (maximum disagreement), $d_C(c) = 0$ if the all committee agrees on the processing of *c* (minimum disagreement). Note that we quantify disagreement based on the class chosen by each candidate solution in *C* (extracted vs. not extracted) [27] without any reference to forms of confidence value, margin or probability [23, 28]. As we pointed out already in the introduction, such notions are not made available by the solver that we have chosen to use.

The procedure for constructing candidate queries takes a set of regular expressions *C* as parameter and determines the character *c* $\in t$ with maximal disagreement $d_C(c)$ in the full input set *t*. Next, the procedure determines the set *S* of candidate queries as the set composed of all snippets of *t* which meet the following conditions: (a) are extracted by at least a regular expression in *C*, (b) overlap *c* ′, and (c) do not overlap any available annotation.

We implemented two variants of a query builder. The *Query by committee* (QbC) variant works as follows: (a) construct the set *S* of candidate queries using the full population as committee *C*, (b) compute, for each snippet in *S*, the average disagreement among the characters of the snippet, and (c) choose the snippet with minimal average disagreement as query. The *Query by restricted committee* (rQbC) variant is similar to QbC except that the committee *C* contains only the best 25% of the current population (ranking being based on the current set of matches and unmatches).

QbC and rQbC are based on a principle widely used in active learning [31, 33], i.e., on the assumption that the query for which an ensemble of competing hypotheses exhibits maximal disagreement is the most informative for the learning task [34]. Such a principle has been used also in active learning for Genetic Programming [13, 30, 19]—in those scenarios there is the problem of choosing a candidate query but not the one of constructing queries, though. Indeed, the proposal in [19] augments this principle by also taking into account a measure of diversity between each candidate query and queries already answered. Our preliminary exploration of this additional principle, that we do not illustrate for space reasons, has not delivered satisfactory results. We believe the reason consists in the difficulty of finding a diversity measure for text snippets correlated with diversity between regular expressions—e.g., two text snippets could be very different while at the same time they could be captured by the same regular expression or by regular expressions that are very similar.

Concerning query builders we also observe that a wealth of active learning approaches choose queries based on uncertainty of the current solution, especially when the learner is not based on an ensemble of competing hypotheses [23, 32, 31, 35]. On the other hand, such approaches do not fit the state-of-the-art regex learner that we use in our system, because such a learner does not provide any confidence level about the handling of a given snippet (i.e., extracted vs. not extracted) by the current solution.

We also implemented a third query builder that randomly chooses an unannotated snippet. We place an upper bound to the maximum length of the query that may be generated: we set the actual bound value in our experimental evaluation to the maximum size of a desired extraction across all our datasets (few hundreds characters). The upper bound causes this query builder to filter out candidate queries which are too long, which hence advantages this builder w.r.t. one which selects a truly random snippet of *t*. For this reason, we call this builder SmartRand.

3. EXPERIMENTS

We focused on the *extraction performance* of the regular expression generated for a given amount of *user annotation effort*. We quantify extraction performance with F-measure (Fm), which is the harmonic mean of precision (ratio between the number of correctly extracted snippets and the number of all the extracted snippets) and recall (ratio between the number of correctly extracted snippets and the number of all the snippets which should have been extracted). We chose to quantify user annotation effort by the number of annotated characters (AC).

We evaluated all the 9 combinations between the proposed design variants and we considered 11 challenging extraction tasks used in [6]. For each extraction task, we randomly selected a subset of the original corpus containing approximately 100 desired extractions. The name of each extraction task can be seen—along with the size of the input text expressed in number of characters—in Table 1: it is composed of the name of the corpus followed by the name of the entity type to be extracted.

We assessed our system variants as follows. For each task and variant, we chose a random desired extraction as the only starting annotated snippet and executed the variant with a simulated oracle. We repeated the above procedure 15 times, with 5 different starting matches and 3 different random seeds. We terminated each execution upon the query for which either at least 25% of the available characters was annotated or the F-measure on the full input text (i.e., not only on the annotated portion) was 1. Although a real deployment cannot quantify F-measure on a yet unannotated input text, we chose to include the latter condition in the termination criterion in order to provide a fair assessment of variants which are able to generate perfect solutions before reaching the predefined annotation budget. We chose 25% of the available characters as annotation budget because we have found that, with these datasets, it corresponds to a few minutes of actual annotation.
Table 1 shows the main results (statistical significance is analyzed in more detail later). For each task, Fm is computed on the full input text and averaged across the 15 repetitions of each experiment. Values in the bottom rows of the table are averaged across all tasks. We define the computational effort (CE) as the number of characters analyzed for fitness evaluations across an execution. This quantity is a hardware-independent performance index. Execution times are in the order of minutes, similarly to [6], we do not list them in detail for space reasons: the time taken by the query trigger and the query builder is negligible w.r.t. the time taken by the solver.

It can be seen that for nearly all tasks several of our active learning variants are able to generate regular expressions of very good quality. This result is significant because it strongly suggests that active learning is indeed a viable framework for the task of automatic generation of regular expressions.

Another important outcome is that the rQbC query builder tends to deliver better F-measure than the SmartRand query builder while requiring less annotations—ΔFm between 0.05 and 0.1 on the average. In many applications of active learning, a random query chooser is often quite effective and often turns out to be a challenging baseline for more sophisticated query choice strategies [2, 18, 35]. Although we may observe this phenomenon also in our scenario (in which the random selection is enhanced by a length-based filtering, see Section 2), we also observe a clear superiority of approaches based on rQbC. The QbC query builder, on the other hand, is not effective as it tends to exhibit worse results from the three points of view summarized in the table: F-measure, annotation effort, computational effort.

We speculate that the superiority of rQbC over SmartRand may become even more apparent with datasets in which the density of desired extractions is smaller than ours—in our datasets, the likelihood of randomly choosing a snippet that partly overlaps a desired extraction is not very small. We need to investigate this conjecture further, however.

Concerning the behavior of query triggers with rQbC, it can be seen that each of the three options analyzed belongs to a different region of the design space. The Const30 query trigger is much faster (CE) at the expense of obtaining a relatively good but smaller F-measure, while at the same time requiring more annotations (AC). Const200 and Solved represent more useful trade-offs because they deliver the best average F-measure: they require the same amount of annotations, trading a small difference in F-measure for a substantial difference in computational effort.

In order to illustrate the significance of these results further, we executed the state-of-the-art learner proposed in [6] on the same tasks. This learner requires a training set fully annotated before starting execution. For each task we randomly generated 5 training sets, each one with 25% of the available characters and with a random generation procedure carefully tailored so as to ensure that each training set contains approximately 25% of the desired extractions. It may be useful to emphasize that the size of the training set corresponds to the size of the training set of active learning upon the last query: in this case the training set is instead available to the solver for the full execution; furthermore, in active learning the user need not take any effort to dig out an adequate amount of desired extractions from the (potentially large) available data. We executed each task 5 times, each execution using one of the 5 different training sets. We obtained, on average, Fm = 0.97, CE = 29.8 × 10⁶ and AC = 3748, i.e., 49% more annotated characters than rQbC-Const200 and 48% more than rQbC-Solved.

We performed an analysis of the statistical significance of the results based on the Wilcoxon signed-rank test: we chose this test since it is non-parametric and does not require the population to be normally distributed. The results are in Table 2a (F-measure) and 2b (annotated characters, AC)—we omit results about CE for space reasons. In each table, cell (i,j) contains the difference in the average value of the corresponding performance index between variant in row i and variant in row j. Statistical significance of performance index comparison is indicated for varying p-values of the test and highlighted with asterisks.

These results confirm the analysis of Table 1, but they also indicate that the rQbC/Const200 and rQbC/Solved does not guarantee any statistically significant improvement in Fm over SmartRand/Const200. On the other hand, there is indeed some statistically significant evidence of an improvement in terms of smaller annotation effort—12.5% for rQbC/Const200 and 11.8% for rQbC/Solved. Concerning
5. REFERENCES


<table>
<thead>
<tr>
<th>Variant</th>
<th>QbC/Const30</th>
<th>QbC/Const200</th>
<th>QbC/Solved</th>
<th>SmartRand/Const30</th>
<th>SmartRand/Const200</th>
<th>SmartRand/Solved</th>
<th>rQbC/Const30</th>
<th>rQbC/Const200</th>
<th>rQbC/Solved</th>
<th>rQbC/Const30</th>
<th>rQbC/Const200</th>
<th>rQbC/Solved</th>
</tr>
</thead>
<tbody>
<tr>
<td>QbC/Const30</td>
<td>0.03***</td>
<td>-0.03***</td>
<td>0.01</td>
<td>-0.07***</td>
<td>-0.03***</td>
<td>0.01</td>
<td>-0.07***</td>
<td>-0.03***</td>
<td>0.01</td>
<td>-0.07***</td>
<td>-0.03***</td>
<td>0.01</td>
</tr>
<tr>
<td>QbC/Const200</td>
<td>0.05***</td>
<td>0.02*</td>
<td>0.06***</td>
<td>-0.02</td>
<td>-0.04***</td>
<td>0.02</td>
<td>0.05***</td>
<td>-0.02***</td>
<td>0.01</td>
<td>-0.04***</td>
<td>-0.03***</td>
<td>0.01</td>
</tr>
<tr>
<td>SmartRand/Const30</td>
<td>-0.01</td>
<td>-0.04**</td>
<td>-0.06***</td>
<td>-0.08***</td>
<td>-0.04***</td>
<td>-0.01</td>
<td>-0.08***</td>
<td>-0.07***</td>
<td>-0.01</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SmartRand/Const200</td>
<td>0.07***</td>
<td>0.04**</td>
<td>0.02</td>
<td>0.08***</td>
<td>0.04***</td>
<td>0.07***</td>
<td>0.00</td>
<td>0.01</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SmartRand/Solved</td>
<td>0.03***</td>
<td>0.00</td>
<td>0.04***</td>
<td>-0.04***</td>
<td>0.03***</td>
<td>-0.04***</td>
<td>0.03***</td>
<td>0.03***</td>
<td>0.01</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rQbC/Const30</td>
<td>-0.01*</td>
<td>-0.03***</td>
<td>-0.05***</td>
<td>0.01</td>
<td>-0.07***</td>
<td>-0.03***</td>
<td>0.07***</td>
<td>0.01</td>
<td>-0.06*</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rQbC/Const200</td>
<td>0.07***</td>
<td>0.04**</td>
<td>0.02</td>
<td>0.08***</td>
<td>0.00</td>
<td>0.04***</td>
<td>0.07***</td>
<td>0.01</td>
<td>-0.06**</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rQbC/Solved</td>
<td>0.05***</td>
<td>0.03***</td>
<td>0.01***</td>
<td>0.07***</td>
<td>-0.01</td>
<td>0.03***</td>
<td>0.06***</td>
<td>-0.01</td>
<td>0.01</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Average differences of Fm and AC of pairs of the proposed variants. For each pair, the statistical significance is shown: *: p < 0.1, **: p < 0.05, ***: p < 0.01 (the last condition corresponds to the strongest statistical significance; absence of any asterisk indicates that the comparison is not statistically significant, i.e., p ≥ 0.1).


