Syntactical Similarity Learning by means of Grammatical Evolution

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

Department of Engineering and Architecture, University of Trieste, Trieste, Italy

Abstract. Several research efforts have shown that a similarity function synthesized from examples may capture an application-specific similarity criterion in a way that fits the application needs more effectively than a generic distance definition. In this work, we propose a similarity learning algorithm tailored to problems of syntax-based entity extraction from unstructured text streams. The algorithm takes in input pairs of strings along with an indication of whether they adhere or not adhere to the same syntactic pattern. Our approach is based on Grammatical Evolution and explores systematically a similarity definition space including all functions that may be expressed with a specialized, simple language that we have defined for this purpose. We assessed our proposal on patterns representative of practical applications. The results suggest that the proposed approach is indeed feasible and that the learned similarity function is more effective than the Levenshtein distance and the Jaccard similarity index.

Keywords: distance learning, entity extraction, string patterns

1 Introduction and related work

Many solutions to practically relevant applications are based on techniques that rely on a form of *similarity* between data items, i.e., on a quantification of the difference between any pair of data items in a given feature space. Although such a similarity may be quantified by many different generic functions, i.e., distances or pseudo-distances, a wealth of research efforts have advocated the usage of similarity functions that are *learned* from collections of data pairs labelled as being either "similar" or "dissimilar" [1–3]. Indeed, similarity functions constructed by a *similarity learning* algorithm have proven very powerful in many different application domains, as such functions may capture the application-specific similarity criterion described by the available *examples* in a way that fits the application needs more effectively than a generic distance definition.

In this work, we focus on the problem of learning a similarity function suitable for syntax-based *entity extraction* from *unstructured text* streams. The identification of strings which adhere to a certain syntactic pattern is an essential component of many workflows leveraging digital data and such a task occurs routinely in virtually every sector of business, government, science, technology.

Devising a similarity function capable of capturing syntactic patterns is an important problem as it may enable significant improvements in methods for constructing syntax-based entity extractors from examples automatically [4–14]. We are not aware of any similarity definition capable of (approximately) separating strings which adhere to a common syntactic pattern (e.g., telephone numbers, or email addresses) from strings which do not.

We propose an approach based on GE, in which we explore systematically a similarity definition space including all functions that may be expressed with a specialized, simple *language* that we have defined for this purpose. The language includes the basic flow control, arithmetic and relation operators. It is expressive enough to describe important, existing similarity definitions, that we use as baseline in our experimental evaluation. A candidate solution, i.e., an individual, represents a program in the language which takes a pair of strings as input and outputs a number quantifying their similarity. Programs are executed with a *virtual machine* that we designed and implemented. The virtual machine is necessary only for assessing the quality of candidate solutions during the evolutionary search: the final solution can obviously be implemented in a more compact and more efficient way based on the specific technology in which the learned similarity function will be inserted.

We assessed our proposal on several tasks representative of practical applications, each task being a large text stream annotated with the strings following a task-specific pattern. We emphasize that we did not learn one similarity definition for each task: instead, we learned a single similarity function from all tasks except for one and then evaluated the behavior of the learned similarity function on the remaining task—i.e., on a syntactic pattern that was not available while learning. The results, averaged across all the tasks, demonstrate that the proposed approach is indeed feasible, i.e., it is able to learn a similarity function capable of (approximately) separating strings based on their adherence to a given syntactical pattern. Most importantly, the learned function is more effective than the Levenshtein distance and the Jaccard similarity index.

An evolutionary approach to metric learning can be found in [15]. The cited work proposes a general approach for multi-label clustering problems in a given feature space. We focus instead on a different and more specific problem: syntax-based entity extraction from unstructured text streams. Furthermore, we aim at learning a similarity function and do not insist in requiring that the learned function be a distance. Several proposals have advocated genetic approaches to similarity learning in the context of case-based reasoning [16–18]. In those cases, though, the problem was learning a meaningful similarity criterion between problem definitions, to enable effective comparison of a new problem to a library of known, already solved problems. We consider instead similarity between pairs of strings that are a small part of a problem instance. Our problem statement follows a common approach in similarity learning: input data consist of pairs of data points, where each pair is known to belong to either the same class (i.e., the same pattern) or to different classes [1]. An alternative framework is based on input data which consist of triplets of data points (a, b, c) labelled with the information regarding whether a is more similar to b or to c [19–21]. Such a *relative comparisons* framework has proven to be quite powerful, in particular, for clustering applications. A relative comparison approach could be applied also to our entity extraction problem and indeed deserves further investigation.

1.1 Problem statement

The problem input consists of a set of tasks $\{T_1, \dots, T_n\}$ where each task describes a *syntactic pattern* by means of *examples*. Task T_i consists of a pair of sets of strings (P_i, N_i): P_i contains strings which adhere to the ith pattern while N_i contains strings which do not adhere to that pattern. The problem consists in learning a *similarity function* $\hat{m}(s, s')$ which, given two strings s, s', returns a *similarity index* capable of capturing to which degree s and s' adhere to the same (unknown) syntactic pattern. That is, intuitively, pairs of strings in P_i should be associated with a "large" similarity index, while pairs consisting of a string in P_i and a string in N_i should be associated with a "small" similarity index. Furthermore, this requirement should be satisfied for all tasks by the same function $\hat{m}$.

In details, the ideal learned function should satisfy the following requirement:

$$\forall i \in \{1, \dots, n\}, \forall x \in M(P_i, N_i), \forall y \in M(P_i, P_i), x < y \tag{1}$$

where $M(S, S') = \{m(s, s') : s \in S, s \in S'\}$. For a given problem input, a function satisfying Equation 1 may or may not exist; and, even if it exists, a learning algorithm may or may not be capable of learning that function.

2 Our approach

2.1 Search space and solution quality

We consider a search space composed of functions that may be expressed with the language L described in Figure 1 in the Backus-Naur Form (BNF). The available mathematical operators are defined in the rule concerning the ⟨ValueReturningFunction⟩ non-terminal while relation operators are defined in rule concerning the ⟨Condition⟩ non-terminal. The language includes basic flow control operators and allows defining numeric variables and arrays dynamically. Access to variables and array elements occur by index.

The language is expressive enough to describe commonly used similarity indexes: in particular, we described the Levenshtein distance and the Jaccard similarity index—which we used in our experimental evaluation as baselines—using this language.

We propose an evolutionary approach based on *Grammatical Evolution* (GE) [22, 23]. GE is an evolutionary framework where candidate solutions (*individuals*) are represented as fixed-length numeric sequences. Such sequences (*genotype*) are translated into similarity functions (*phenotype*) by means of a mapping procedure which uses the production rules in a grammar definition.

Rules

```
 1. ⟨BlockCode⟩ ::= ⟨RowOfBlockCode⟩
 2. ⟨Statement⟩ ::= ⟨Assign⟩ | ⟨CreateArray⟩ | ⟨CreateVariable⟩ | ⟨For⟩ | ⟨If⟩ |
    ⟨Return⟩ | ⟨SetArrayItem⟩
 3. ⟨ValueReturningFunction⟩ ::=⟨Constant⟩ | ⟨GetVariableValue⟩ | ⟨Add⟩ | ⟨Decrement⟩ |
    ⟨Maximum⟩ | ⟨Minimum⟩ | ⟨GetArrayItem⟩ | ⟨GetArrayLength⟩ | ⟨Division⟩ |
    ⟨Multiplication⟩
 4  ⟨Assign⟩ ::= var[⟨ValueReturningFunction⟩] = ⟨ValueReturningFunction⟩
 5. ⟨CreateArray⟩ ::= newArray[⟨ValueReturningFunction⟩]
 6. ⟨CreateVariable⟩ ::= createVariable()
 7. ⟨Division⟩ ::= (⟨ValueReturningFunction⟩ / ⟨ValueReturningFunction⟩)
 8. ⟨For⟩ ::= for(index0 = 0; index0 < ⟨ValueReturningFunction⟩; index0++) ⟨BlockCode⟩
 9. ⟨If⟩ ::= if(⟨Condition⟩) ⟨BlockCode⟩ else ⟨BlockCode⟩
10. ⟨Return⟩ ::= return ⟨ValueReturningFunction⟩
11. ⟨SetArrayItem⟩ ::= array[⟨ValueReturningFunction⟩][⟨ValueReturningFunction⟩] =
    ⟨ValueReturningFunction⟩
12. ⟨Add⟩ ::= ⟨ValueReturningFunction⟩ + ⟨ValueReturningFunction⟩
13. ⟨Subtract⟩ ::= ⟨ValueReturningFunction⟩ - ⟨ValueReturningFunction⟩
14. ⟨Maximum⟩ ::= maximum(⟨ValueReturningFunction⟩,⟨ValueReturningFunction⟩)
15. ⟨Minimum⟩ ::= minimum(⟨ValueReturningFunction⟩,⟨ValueReturningFunction⟩)
16. ⟨Multiplication⟩ ::= ⟨ValueReturningFunction⟩ * ⟨ValueReturningFunction⟩
17. ⟨GetArrayItem⟩ ::= array[⟨ValueReturningFunction⟩][⟨ValueReturningFunction⟩]
18. ⟨GetArrayLength⟩ ::= array[⟨ValueReturningFunction⟩].length
19. ⟨Constant⟩ ::= 0 | 1 | ... | 255
20. ⟨GetVariableValue⟩ ::= var[⟨ValueReturningFunction⟩]
21. ⟨RowOfBlockCode⟩ ::= ⟨Statement⟩ | ⟨Statement⟩ \n ⟨RowOfBlockCode⟩
22. ⟨Condition⟩ ::= ⟨EqualCondition⟩ | ⟨NotEqualCondition⟩ |
    ⟨GreaterCondition⟩ | ⟨GreaterOrEqualCondition⟩
23. ⟨EqualCondition⟩ ::= ⟨ValueReturningFunction⟩ == ⟨ValueReturningFunction⟩
24. ⟨NotEqualCondition⟩ ::= ⟨ValueReturningFunction⟩ != ⟨ValueReturningFunction⟩
25. ⟨GreaterCondition⟩ ::= ⟨ValueReturningFunction⟩ > ⟨ValueReturningFunction⟩
26. ⟨GreaterOrEqualCondition⟩ ::= ⟨ValueReturningFunction⟩ >= ⟨ValueReturningFunction⟩
```

Alternative rules

```
 2. ⟨Statement⟩ ::= ⟨CreateVariable⟩
 3. ⟨ValueReturningFunction⟩ ::= ⟨Constant⟩
21. ⟨RowOfBlockCode⟩ ::= ⟨Statement⟩
```

Fig. 1. BNF grammar for the language *L*: below the set of alternative rules (see text).

After early experimentation, we chose to tailor several aspects of the general GE framework to our specific problem.

In our case, we represent an individual with a genotype consisting of a tuple $\boldsymbol{g} \in [0, 255]^{n_\text{gen}}$, where each g_i element is a positive 8-bit integer. We chose $n_\text{gen} = 350$ because with such value we were able to obtain, from two suitable genotypes, the phenotypes corresponding to the Levenshtein distance and the Jaccard similarity, according to the mapping procedure described below. Given a genotype, we obtain the corresponding phenotype, i.e., a similarity function expressed as a program l in the language L, according to an iterative *mapping procedure* which works as follows, starting with $l = \langle\text{BlockCode}\rangle$ and $i = 0$: (i) we consider the first occurrence of a non-terminal in l and the corresponding rule in the BNF grammar for L; (ii) among the $n_\text{rule} \geq 1$ alternatives (i.e., possible replacements separated by | in the rule), we choose the $(j + 1)$th one, with j equals to the remainder between g_i and n_rule; (iii) we increment i by one: if i exceeds n_gen, we set to 1. The procedure is iterated until no more non-terminals exist in l: since it is not guaranteed that this condition is satisfied in a finite number of iterations, we implemented a mechanism to overcome this limitation. We associate a number c with each non-terminal x in l: the value of c is set to 0 for the starting non-terminal $\langle\text{BlockCode}\rangle$, or to $c' + 1$ otherwise, where c' is the number associated with the non-terminal whose replacement lead to the insertion of x in l. Whenever a non-terminal among $\langle\text{Statement}\rangle$, $\langle\text{ValueReturningFunction}\rangle$, and $\langle\text{RowOfBlockCode}\rangle$ has to be replaced, if its c exceeds a parameter $c_\text{max} = 40$, we use the alternative rules shown at the bottom of Figure 1 instead of the original ones for those non-terminals—in other words, with this mechanism we pose a depth limit on the derivation trees.

We quantify the quality of an individual encoding a similarity function m by its *fitness* $f(m)$, that we define as follows. Given a numeric multiset I, let $I_{p\%}$ indicate the smallest element $i \in I$ greater or equal to the p percentile of elements in I. Given a pair of numeric multisets (X, Y), we define the *overlapness* function $o(X, Y; p) \in [0, 1]$ as follows:

$$o(X, Y; p) = \frac{|\{x \in X : x \geq Y_{p\%}\}| + |\{y \in Y : y \leq X_{(100-p)\%}\}|}{|X| + |Y|} \tag{2}$$

Intuitively, $o(X, Y; p)$ measures the degree of overlapping between elements of X and Y, assuming that elements in X are in general smaller than elements of Y: when X and Y are perfectly separated, $o(X, Y; p) = 0, \forall p$. The value of p is used to discard extreme (greatest for X and smallest for Y) elements in the multisets. The fitness $f(m) \in [0, 1]$ of m is given by:

$$f(m) = \frac{1}{2n} \sum_{i=1}^{n} o\big(M(P_i, N_i), M(P_i, P_i); 10\big) + o\big(M(P_i, N_i), M(P_i, P_i); 0\big) \tag{3}$$

where $M(S, S')$ is defined as for Equation 1. In other words, the fitness of m is the average overlapness over the tasks in $\{T_1, \dots, T_n\}$: for each task, $f(m)$ takes into account the average between the overlapness of the two multisets

$M(P_i, N_i)$ and $M(P_i, P_i)$ computed on the whole multisets and after discarding 10% extreme values. The rationale for the latter design choice is to avoid giving too much importance to possible outliers in the data. Note that a similarity function satisfying Equation 1 has zero fitness—i.e., fitness should be minimized.

During the evolutionary search, we evolve a fixed-size population of n_pop individuals for $n_\text{iter} = 200$ generations by means of the *mutation* and *two-point crossover* genetic operators, which are applied to individuals selected by means of a tournament of size 3.

2.2 Virtual Machine

We designed and implemented a *virtual machine* (VM) capable of executing programs in language L. A VM program execution takes a pair of strings (s, s') as input and returns the value $m(s, s')$, m being the similarity function represented by the program.

As described in Section 2.1, the language allows defining numeric variables and arrays dynamically with access occurring by index. VM provides a running program with a list of numeric variables and a list of numeric arrays. Indexes start from 0 and when a new variable is created the next free index is used: the actual variable/array being accessed is determined by the reminder of $\frac{i}{n_v}$. When execution starts, VM creates two arrays into the arrays list, one for s and the other for s': the ith element of each array contains the UTF-8 representation of the ith character in the corresponding string. The execution stops when a return statement is reached or when the last instruction has been executed: in the latter case, the returned value is $m(s, s') = 0$.

A VM program execution may fail, in which case execution terminates and the returned value is $m(s, s') = 0$. Failure occurs when one of the following conditions is met: division by zero; maximum number n_max of executed instructions exceeded; maximum array size n_array exceeded—we set $n_\text{max} = 40\,000$ and $n_\text{max} = 10 \, \text{length}(s) \, \text{length}(s')$.

3 Experimental evaluation

As described previously, a task describes a syntactic pattern by means of examples, i.e., each task consists of a pair of sets of strings (P_i, N_i): P_i contains strings which adhere to the pattern while N_i contains strings which do not adhere to the pattern. We assess our proposal on several datasets representative of possible applications of our similarity learning method (the name of each dataset describes the nature of the data and the type of the entities to be extracted): HTML-href [14, 13, 11], Log-MAC+IP [14, 13, 11], Email-Phone [14, 13, 11, 8, 7], Bills-Date [14, 12], Web-URL [14, 13, 11, 7], Twitter-URL [14, 13, 11]. Each dataset consists of a text annotated with all and only the snippets that should be extracted.

We constructed a task (P, N) for each such dataset, as follows. Let d denote the annotated text in the dataset. Set P contains all and only the strings that should be extracted from d. Set N contains strings obtained by splitting the remaining part of d. It follows that no pair of elements in $P \cup N$ overlap. The splitting procedure is based on a *tokenization* heuristics that (approximately) identifies the tokens that delimit P strings in d; those tokens are then used for splitting N strings in d as well. For example, if strings in P are delimited by a space, then we split the remaining part of d by spaces and insert all the resulting strings in N. The details of the heuristic are complex because different P strings could be delimited by different characters—we omit the details for ease of presentation.

We performed a cross-fold assessment of our proposed method, i.e., we executed one experiment for each of the 6 tasks resulting from the available datasets. In each ith experiment we executed our method on a *learning set* consisting of all but the ith task. We obtained the actual jth pair (P'_j, N'_j) of the learning set by sampling $2n_\text{ex}$ items of the corresponding (P_j, N_j), i.e., $|P'_j| = |N'_j| = n_\text{ex}$, with $P'_j \subseteq P_j$, $N'_j \subseteq N_j$, where n_ex is a parameter of the experiment which affects the amount of data available for learning.

We used the remaining task (P_i, N'_i) (i.e., all of the examples in P_i and a number $|N'_i| = |P_i|$ of examples sampled randomly from N_i) for quantifying the quality of the learned similarity function $m^\star$—$m^\star$ being the individual with the best fitness after the last generation. Note that we assessed $m^\star$ on a task *different* from the tasks that we used for learning it.

For each task, we repeated the experiment for 5 times, each time using a different random seed. We considered the following indexes for each experiment, which we averaged across the 5 repetitions: the learning fitness LF, i.e., the fitness of $m^\star$ on the learning set; the testing fitness TF, i.e., the fitness of $m^\star$ on (P_i, N'_i); the number #I of instructions in $m^\star$; the average number #S of executed instructions while processing pairs in (P_i, N'_i) with $m^\star$.

We explored two different values for the population size n_pop, 50 and 100 individuals, and three different values for the cardinality of sets of examples n_ex: 10, 25 and 50.

Table 1 provides the key results (with $n_\text{ex} = 50$ and $n_\text{pop} = 50$), separately for each dataset and averaged across all datasets. To place results in perspective, we provide all indexes (except for LF) also for two baseline definitions: the Levenshtein distance, which counts the minimum number of character insertions, replacements or deletions required to change one string into the other, and the Jaccard similarity index, which considers each string as a set of bigrams and is the ratio between the intersection and the union of the two sets. The key result is that, on average, the definitions synthesized by our method exhibit the best results. By looking at individual tasks, our synthesized definitions outperform Jaccard in three tasks, are nearly equivalent in one task and are worse or slightly worse in the two remaining tasks. Thus, the similarity functions synthesized by our method are more effective at separating strings based on their adherence at a certain syntactic pattern with respect to the traditional Levenshtein and Jaccard metrics.

Table 2 provides further insights into our method by providing results averaged across all tasks for various combinations of available examples n_ex and population

Table 1. Results of our method, with $n_{\mathrm{ex}} = 50$ and $n_{\mathrm{pop}} = 50$, and the baselines. Best TF figure highlighted.

Task	LF	TF			#I			#S [$\times 10^6$]		
	GE	GE	Jac.	Lev.	GE	Jac.	Lev.	GE	Jac.	Lev.
HTML-href	0.45	**0.42**	0.64	0.91	1877	174	103	0.22	3.49	2.25
Log-MAC+IP	0.44	**0.08**	0.82	0.91	179	174	103	0.06	0.42	0.75
Email-Phone	0.43	0.64	**0.56**	0.90	352	174	103	0.41	4.62	3.64
Bills-Date	0.49	0.85	**0.59**	0.90	1116	174	103	1.56	2.71	5.19
Web-URL	0.40	**0.30**	0.43	0.92	151	174	103	0.72	23.8	10.00
Twitter-URL	0.48	0.30	**0.29**	0.90	147	174	103	0.84	6.28	8.10
Average	0.45	**0.43**	0.55	0.90	637	174	103	0.64	6.90	4.99

size n_{pop}. It can be seen that, with a larger population ($n_{\mathrm{pop}} = 100$), the amount of learning examples does not impact TF significantly, but more examples lead to more compact and more efficient solutions (smaller #I and #S, respectively). On the other hand, the configuration with smaller population ($n_{\mathrm{pop}} = 50$) exhibits a slight but consistent improvement in TF when the amount of examples grows. It can also be observed that more examples lead to solutions with varying length but that tend to be more efficient (no clear trend in #I and decreasing #S, respectively). This observation suggests that our method might perhaps be improved further by a multiobjective optimization search strategy, where the fitness of an individual would take into account not only its ability of capturing similarity as specified in the learning examples (to be maximized) but also the length of the individual (to be minimized).

Table 2. Results (including learning time t_l) for different values of n_{pop} and n_{ex}.

n_{pop}	n_{ex}	LF	TF	#I	#S [$\times 10^6$]	t_l [s]
50	10	0.37	0.45	552	0.59	52
	25	0.43	0.44	3076	0.56	245
	50	0.45	**0.43**	637	0.64	715
100	10	0.34	0.50	1138	2.76	110
	25	0.40	0.48	1224	0.94	326
	50	0.38	0.49	**443**	**0.44**	1056

Table 2 also shows the learning time t_l, averaged across repetition: we performed the experiments on a platform equipped with an Intel Core i7-4720HQ (2.60 GHz) CPU and 16 GB of RAM.

4 Concluding remarks

We have investigated the feasibility of learning a similarity function capable of (approximately) separating strings which adhere to a common syntactic pattern (e.g., telephone numbers, or email addresses) from strings which do not. We are not aware of any similarity function with this property, which could enable significant improvements in methods for constructing syntax-based entity extractors from examples automatically—in many application domains, similarity functions learned over labelled sets of data points have often proven more effective than generic distance definitions.

We have proposed a method based on Grammatical Evolution which takes pairs of strings as input, along with an indication of whether they follow a similar syntactic pattern. The method synthesizes a similarity function expressed in a specialized, simple language that we have defined for this purpose.

We assessed our proposal on several tasks representative of practical applications, with an experimental protocol in which we learned a similarity function on a given set of tasks (i.e., patterns) and we assessed the learned function on a previously unseen task. The results demonstrate that the proposed approach is indeed feasible and that the learned similarity function is much more effective than the Levenshtein distance and the Jaccard similarity index.

We plan to extend our investigation in two ways: first, synthesize a more powerful similarity function, by using a broader set of patterns and a larger amount of labelled data points; in this phase there may certainly be room for further improvements to our Grammatical Evolution method; next, take advantage of the learned similarity function in order to improve methods for syntax-based entity extraction.

Acknowledgements

We are grateful to Michele Furlanetto who contributed in the implementation of our proposed method.

References

1. Yang, L., Jin, R.: Distance metric learning: A comprehensive survey. Michigan State Universiy **2** (2006)
2. Kulis, B.: Metric learning: A survey. Foundations and Trends in Machine Learning **5**(4) (2012) 287–364
3. Bellet, A., Habrard, A., Sebban, M.: A survey on metric learning for feature vectors and structured data. arXiv preprint arXiv:1306.6709 (2013)
4. Fernau, H.: Algorithms for learning regular expressions from positive data. Information and Computation **207**(4) (2009) 521 – 541
5. Cicchello, O., Kremer, S.C.: Inducing grammars from sparse data sets: a survey of algorithms and results. The Journal of Machine Learning Research **4** (2003) 603–632

6. Cetinkaya, A.: Regular expression generation through grammatical evolution. In: Proceedings of the 2007 GECCO conference companion on Genetic and evolutionary computation, ACM (2007) 2643–2646
7. Li, Y., Krishnamurthy, R., Raghavan, S., Vaithyanathan, S., Jagadish, H.: Regular expression learning for information extraction. In: Proceedings of the Conference on Empirical Methods in Natural Language Processing, Association for Computational Linguistics (2008) 21–30
8. Brauer, F., Rieger, R., Mocan, A., Barczynski, W.M.: 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) 1285–1294
9. Murthy, K., P., D., Deshpande, P.: Improving recall of regular expressions for information extraction. In: Web Information Systems Engineering - WISE 2012. Volume 7651 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2012) 455–467
10. Bartoli, A., Davanzo, G., De Lorenzo, A., Mauri, M., Medvet, E., Sorio, E.: Automatic generation of regular expressions from examples with genetic programming. In: Proceedings of the 14th annual conference companion on Genetic and evolutionary computation, ACM (2012) 1477–1478
11. Bartoli, A., Davanzo, G., De Lorenzo, A., Medvet, E., Sorio, E.: Automatic synthesis of regular expressions from examples. Computer (12) (2014) 72–80
12. Bartoli, A., De Lorenzo, A., Medvet, E., Tarlao, F.: Learning text patterns using separate-and-conquer genetic programming. In: Genetic Programming. Springer (2015) 16–27
13. Bartoli, A., De Lorenzo, A., Medvet, E., Tarlao, F.: Active learning approaches for learning regular expressions with genetic programming. In: Proceedings of the 31st Annual ACM Symposium on Applied Computing, ACM (2016) 97–102
14. Bartoli, A., De Lorenzo, A., Medvet, E., Tarlao, F.: Inference of regular expressions for text extraction from examples. Knowledge and Data Engineering, IEEE Transactions on **28**(5) (2016) 1217–1230
15. Megano, T., Fukui, K.i., Numao, M., Ono, S.: Evolutionary multi-objective distance metric learning for multi-label clustering. In: Evolutionary Computation (CEC), 2015 IEEE Congress on, IEEE (2015) 2945–2952
16. Stahl, A., Gabel, T.: Using evolution programs to learn local similarity measures. In: Case-Based Reasoning Research and Development. Springer (2003) 537–551
17. Xiong, N., Funk, P.: Building similarity metrics reflecting utility in case-based reasoning. Journal of Intelligent & Fuzzy Systems **17**(4) (2006) 407–416
18. Xiong, N.: Learning fuzzy rules for similarity assessment in case-based reasoning. Expert systems with applications **38**(9) (2011) 10780–10786
19. Schultz, M., Joachims, T.: Learning a distance metric from relative comparisons. Advances in neural information processing systems (NIPS) (2004) 41
20. Xiong, S., Pei, Y., Rosales, R., Fern, X.Z.: Active learning from relative comparisons. Knowledge and Data Engineering, IEEE Transactions on **27**(12) (2015) 3166–3175
21. Hao, S., Zhao, P., Hoi, S.C., Miao, C.: Learning relative similarity from data streams: Active online learning approaches. In: Proceedings of the 24th ACM International on Conference on Information and Knowledge Management, ACM (2015) 1181–1190
22. Ryan, C., Collins, J., Neill, M.O.: Grammatical evolution: Evolving programs for an arbitrary language. In: Genetic Programming. Springer (1998) 83–96
23. O'Neill, M., Ryan, C.: Grammatical evolution. IEEE Transactions on Evolutionary Computation **5**(4) (2001) 349–358