

Learning Regular Expressions from Noisy Sequences

Ugo Galassi and Attilio Giordana

Dipartimento di Informatica, Università Amedeo Avogadro
Via Bellini 25G, 15100 - Alessandria, Italy

Abstract. The presence of long gaps dramatically increases the difficulty of detecting and characterizing complex events hidden in long sequences. In order to cope with this problem, a learning algorithm based on an abstraction mechanism is proposed: it can infer the general model of complex events from a set of learning sequences. Events are described by means of regular expressions, and the abstraction mechanism is based on the substitution property of regular languages. The induction algorithm proceeds bottom-up, progressively coarsening the sequence granularity, letting correlations between subsequences, separated by long gaps, naturally emerge. Two abstraction operators are defined. The first one detects, and abstracts into non-terminal symbols, regular expressions not containing iterative constructs. The second one detects and abstracts iterated subsequences. By interleaving the two operators, regular expressions in general form may be inferred. Both operators are based on string alignment algorithms taken from bio-informatics. A restricted form of the algorithm has already been outlined in previous papers, where the emphasis was on applications. Here, the algorithm, in an extended version, is described and analyzed into details.

1 Introduction

Very long discrete sequences are found in many challenging applications of data mining, ranging from DNA analysis to user profiling, and anti-intrusion systems. In most cases this kind of sequences are characterized by sparseness, i.e., short consecutive chains of atomic events (episodes) are interleaved with gaps, where irrelevant facts, or facts related to spurious activities, may occur. We define a partially ordered group of interrelated episodes a *complex event* (CE).

This paper addresses the task of discovering CEs in discrete sequences. The task is made more difficult by assuming the presence of noise, making CEs harder to recognize. Episodes are represented as strings of symbols, being a symbol the label assigned to an atomic event. Moreover, it is assumed that noise can be modeled as insertion, deletion and substitution errors, according to a common practice followed in Pattern Recognition.

Here, regular expressions, extended with attributes [9], are proposed to describe the structure of CEs. Attributes are used to set constraints on atomic events. Therefore, the problem of discovering CE's structure is turned into the

problem of learning regular expressions from sequences containing gaps and noise. The problem of inferring regular grammars from data has been previously investigated by many authors with approaches ranging from computational learning theory [1, 18, 16, 17, 4] to neural networks [6], syntactic pattern recognition [10, 19], and probabilistic automata [8]. Nevertheless, the problem considered here does not match immediately any one of the problems solved by the mentioned approaches. In fact, the task is more complex, because the sentences of the language to learn are hidden inside sequences containing a possibly large amount of irrelevant knowledge, which must be discarded.

In a previous approach [2] a Hierarchical Hidden Markov Model [7, 22, 14, 21] has been proposed to describe CEs. Here, we prefer to distinguish the problem of learning structural properties of a CE from the problem of detecting it in presence of noise and other irrelevant facts. Therefore, an extension of the algorithm by Botta et al. [2] is presented, which is more powerful and is no more bound to HHMMs.

By exploiting properties inherent to regular expressions, an abstraction mechanism has been defined: it allows an event to be seen at different levels of granularity depending on the needs. Such a mechanism is exploited by the learning algorithm, which automatically infers the event descriptions from a database of sequences. An important novelty, with respect to previous works, is a method for detecting and learning recurrent structures inside an event, in presence of noise.

In this paper, the learning algorithm is described in details and an evaluation on artificial data is provided.

2 Learning by Abstraction

The main difficulty in discovering and modeling CEs hidden inside long sequences is due to the presence of long gaps, filled by irrelevant facts, between episodes belonging to a CE. On the one hand, statistical correlations among distant episodes are difficult to detect. On the other hand, the complexity of the *mining* algorithm increases with the length of the portion of sequence to be searched to detect such kind of correlations. The strategy proposed here to cope with such kind of problems is based on an abstraction mechanism.

In AI, abstraction has been proposed by several authors with different acceptations (see [20] for an introduction). The acceptance, adopted here, relies on the property of regular expressions of being closed under substitution [13]: by replacing a subexpression with a new symbol, an abstract expression is obtained. As previously mentioned, CEs are described by means of regular expressions extended with attributes. By applying the substitution property, a CE can be abstracted, or de-abstracted.

The idea will be further clarified describing the scheme of the algorithm used for discovering CEs hidden in a set, \mathcal{LS} , of learning sequences. The algorithm starts bottom-up to construct an abstraction hierarchy, layer after layer. The basic activity at each step consists in identifying episodes occurring with a rele-

vant frequency in \mathcal{LS} : every episode is characterized by a regular expression \mathcal{R} . Then, the detected episodes are *named* by associating a new symbol to each one of them, and episode names become the alphabet for describing \mathcal{LS} at the next abstraction level. Afterwards, every sequence in \mathcal{LS} is abstracted (rewritten) by replacing every episode instance occurring in it with the corresponding episode name. Subsequences of consecutive atomic events, which have not been included in any episode, are replaced with a symbol denoting a *gap*. As it will be described in the next sections, gaps between episodes are considered as a special kind of episodes.

In the new sequences obtained from the abstraction step, episodes, previously separated by subsequences of irrelevant facts, may become consecutive, only separated by one gap symbol. Then, at the next abstraction step, correlations at a wider range can be detected by repeating the same procedure described so far, while the complexity of the algorithm remains affordable.

Important aspects to consider, in order to correctly detect statistical correlations between consecutive atomic events, are the event duration and the distance from one another, which could be required to satisfy specific constraints. As an example, one may be willing to accept a correlation between two events A and B, when B frequently occurs few days after A, but one may want to reject a correlation if the distance of B from A randomly ranges from one day to one year.

The attributes extending regular expressions have principally the function of preserving the information about duration and distance between events through the abstraction process. Every atomic event E is denoted by a name (symbol) and by an attribute l_E reporting the length (duration) of E on the unabstracted sequence. When an episode is abstracted into a new atomic event at the higher level, the length of this last is set to the length of the episode. In the same way, gaps are denoted by a symbol, and have a length set to the distance between the two neighbouring episodes. As it will be described in the following, the event description language allows constraints on duration of an event to be specified. Therefore, to set constraints on the distance between two events is sufficient to set constraints on the gap in between.

This solution, of using gap symbols to fill spaces between non adjacent atomic events, allows for any discrete sequence to be transformed into a *string* of symbols. The important benefit is that a large set of string processing algorithms can be immediately exploited.

3 Regular Expressions

The standard formalism for regular expressions [13] is adopted for describing episodes and CEs. Regular language syntax contains meta-symbols for denoting disjunction and iteration. Disjunction is denoted by the symbol $|$. For instance, the construct $a(c|d)a$ denotes a sequence of three symbols, where the first and the third are "a", and the second may be "c" or "d". Parentheses are used to enclose subexpressions. The special symbol ϵ denotes the null event and is used to model

omission. For instance, expression $a(c|d|\epsilon)a$ entails that also the sentence aa , is a possible event instance. Repetition is denoted by a superscript on a symbol, or on a subexpression, which indicates how many consecutive times it occurs. As an example, expression a^3b^2 is a compact form for denoting the sequence "aaabb".

In principle, regular expressions can also describe infinite sentences. The classical notation for handling infinity consists in using symbol " \star " as a superscript to expressions. Here, infinity is not allowed. Instead, the regular language notation is slightly extended to allow for nondeterminate iterations, where the number of repetitions may range inside a bounded interval. For instance, expression $ab^{3\div 9}$ denotes a sequence whose first element is "a" followed by a number of "b" ranging between 3 to 9.

Constraints on the event/gap length may be set by annotating symbols in regular expressions. Annotation must be included inside square brackets, following the symbol denoting an atomic event. For instance $a[n]$ means that the length l_a of a must be n ($l_a = n$), whereas $a[n, m]$ means that the length of a must range between n and m ($n \leq l_a \leq m$). A legal example of annotation can be as in the following:

$$a[3, 5]^3b[4, 8]^2 \quad (1)$$

Informally, expression (1) specifies that the duration of any event of type a must be in the interval $[3, 5]$ and the duration of any event of type b must be in the interval $[4, 8]$. Gaps are named and annotated as atomic events. However, given the semantics of gaps, iteration has no meaning for them; then, gap names cannot have an exponent.

4 String Alignment and Flexible Matching

A key role in the abstraction process is played by the *approximate matching* of strings and of regular expressions, which, in turn, is based on *string alignment*. String alignment has been deeply investigated in Bio-informatics and a wide collection of effective algorithms are available for doing it[5, 12]. Here some basic concepts, necessary to make the paper self-consistent, will be recalled; the interested reader can find in[5, 12] an exhaustive introduction to the topic.

Definition 1. *Given two strings s_1 and s_2 , let s'_1 and s'_2 be two strings obtained from s_1 and s_2 , respectively, by inserting an arbitrary number of spaces such that the atomic events in the two strings can be put in a one-to-one correspondence. The pair $A(s_1, s_2) = \langle s'_1, s'_2 \rangle$, is said a global alignment between s_1 and s_2 .*

From global alignment, local alignment and multi-alignment can be defined.

Definition 2. *Any global alignment between a pair of substrings r_1 and r_2 extracted from two strings s_1 and s_2 , respectively, is said a local alignment $LA(s_1, s_2)$, between s_1 and s_2 .*

Definition 3. *Given a set S of strings, a multi-alignment $MA(S)$ on S is a set S' of strings, where every string $s \in S$ generates a corresponding string $s' \in S'$*

by inserting a proper number of spaces, and every pair of strings $\langle s'_1, s'_2 \rangle$ is a global alignment $A(s_1, s_2)$ of the corresponding strings s_1, s_2 in set S .

It is immediate to verify that, for a pair of strings s_1 and s_2 , many alignments exist¹. However, the interest is for alignments maximizing (or minimizing) an assigned scoring function². A typical scoring function is string similarity [12], which can be stated in the following general form:

$$f(s_1, s_2) = \sum_{i=1}^n f(s'_1(i), s'_2(i)) \quad (2)$$

being n the length of the alignment $\langle s'_1, s'_2 \rangle$, and $f(\cdot, \cdot)$ a scoring function, which depends upon the symbol pairs, which have been aligned.

An alternative to (2) for aligning strings and estimating similarity is based on a special kind of Hidden Markov Model (HMM) called *profile HMM* (see [5] for an introduction). The fundamental difference between profile HMM and (2) is that for the former the scoring function is stated in terms of a mixture model defining a probability distribution. Then the similarity between two strings s_1 and s_2 , or between a string and a template, is defined as the probability that s_2 be obtained from s_1 as the result of a stochastic sequence of insertions, deletions and substitutions. A profile HMM can be obtained by compilation from a regular expression, as it will be discussed later.

In the framework of Dynamic Programming, the problem of finding an alignment maximizing a similarity function (or a profile HMM) is solvable with complexity of $O(nm)$ being n and m the length of s_1 and s_2 , respectively. Nevertheless, approximate solutions can be found in linear time[12]. On the contrary, the problem of finding an optimal multi-alignment is exponential in the cardinality $|S|$ of set S . Therefore, only approximate solutions can be used when S is large.

The concept of similarity and alignment between strings is easy to extend to the concept of alignment between a string and a regular expression. A regular expression \mathcal{R} is equivalent to a set of strings that can be derived from it. Therefore the optimal alignment between \mathcal{R} and a string s , with respect to an assigned similarity function, is the best alignment among all possible alignments between s and anyone of the strings derivable from \mathcal{R} . In the general case, the complexity for finding such an alignment is $O(nm)$ being m the length of \mathcal{R} and n the length of s [15].

A similar extension holds in the HMM framework, where regular expressions can be translated into HMMs. However, such translation requires the target HMM to be augmented in two ways: (a) in order to deal with the presence of insertion and deletion errors, extra states must be explicitly added; (b) in order to model specific probability distributions, cycles in regular expressions need

¹ If no restriction is set on the possible number of inserted spaces, the number of possible alignments is infinite.

² As *approximate/flexible matching* between two strings, or between a string and a regular expression, is intended the problem of finding the optimal alignment with respect to an assigned scoring function

to be unrolled into a feed-forward graph, where only self-loops are allowed. A description of the problem and of the related methodologies can be found in [5, 2, 11].

A last point to discuss is how constraints, set in regular expressions on event lengths, intervene in the matching procedure. Dealing with such kind of constraints requires only minor changes in the algorithms searching for an optimal alignment: symbols in the input string do not matching the constraints will be considered as insertion errors that do not match any symbols. Consequently, the impact on the final alignment will depend upon the specific scoring function. In a similar way, considering iterated subexpressions, iterations in excess (defect), with respect to the bounds set in the exponent, will be considered as insertion (deletion) errors.

5 The Learning Algorithm

The main learning algorithm includes a basic cycle, activated bottom-up, in which a new abstraction layer is constructed, and a refinement cycle, which can be called top-down one or more times in order to refine the episode descriptions. Both cycles are based on two abstraction operators, ω_S and ω_I , which are used to infer the structure of regular expressions. Operator ω_S constructs regular expressions non containing iterative constructs, whereas ω_I explicitly aims at discovering and abstracting iterative constructs. By interleaving the two operators, an abstraction hierarchy is obtained, from which regular expressions in general form are obtained.

5.1 ω_S Operator

The ω_S operator takes in input a set S of similar substrings, detected using a local alignment algorithm, and constructs an abstract atomic event defined as a pair $\langle \mathcal{R}, E \rangle$ being \mathcal{R} a regular expression generalizing the episode instances contained in S , and E is the abstract event associated to \mathcal{R} . The restriction is that items in \mathcal{R} may be only symbols, or disjunction of symbols. Therefore, no iterative constructs are considered.

The core of ω_S is the construction of the multi-alignment of all strings in the set S ; the similarity measure and the alignment procedure are parameters, which can be assigned according to the needs. The semantics of ω_S consists in the following three step algorithm:

Algorithm ω_S

1. Construct the multi-alignment $MA(S)$ for strings in S .
2. Construct the match graph $MG(S)$.
3. Transform $MG(S)$ into an equivalent regular expression.

The multi-alignment $MA(S)$ is a table whose columns contain the symbols put in correspondence by the alignment algorithm. Therefore, the second step aims at

eliminating noise from episode descriptions preserving possible multi-modalities. Symbols, occurring in a same column more frequently than expected if they would be due to random noise, are considered *match symbols* and will be included in the regular expression generated in the third step. Match symbols are associated to the nodes of a directed graph $MG(S)$. The edges of $MG(S)$ are defined according to the following rule: if there is at least one row in $MA(S)$ where a match symbol x follows a match symbol y , immediately or after one or more spaces, a link from x to y is set in $MG(S)$.

Graph $MG(S)$ is transformed into a regular expression in Step 3. As there are many possible way for doing it, it is not relevant to describe the algorithm into details. In this phase, constraints on the event length (see Section 3) are also learned. We remember that an atomic event E , in a regular expression, can be annotated as $E[n, m]$, being n and m the extremes of the interval in which the event length l_E is accepted. The values for n and m are estimated from the lengths of the instances of E aligned in a same column in $MA(S)$. In this phase, constraints, given a-priori as background knowledge, can also be taken into account.

The algorithm is illustrated through an example in Figure 1, where a regular expression describing a dimorphic occurrence of the word *london*³ in the Italian language is extracted from a set of words affected by typos.

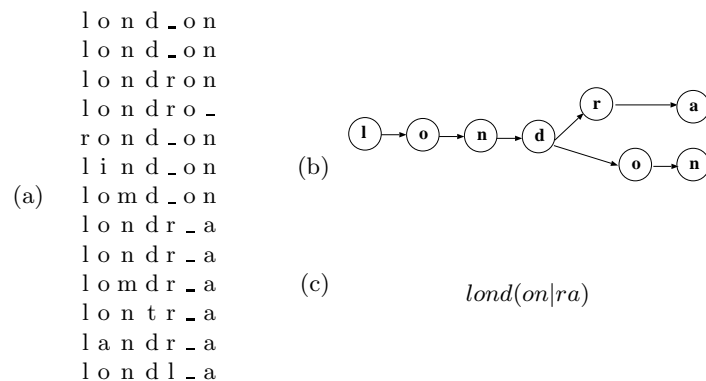


Fig. 1. Example of non-iterative expression obtained from the string set {london, londra, lomdra, lontra, londro, londron, rondon, lindon, london, lomdon, landra, Londra, londla}. (a) Corresponding multi-alignment. (b) Retained alternatives. (c) Final regular expression.

³ Names of foreign towns may occur in an Italian text both in their original orthographic form, or in Italian translation. In this case London is translated into "Londra".

of two different strings is reported in Figure 2 (for the sake of clarity, the entire square matrix has been computed).

5.3 Basic learning cycle

The basic learning cycle consists of four major steps:

1. Non-iterative episode detection. Episodes consisting of non-iterated substrings are detected and abstracted by applying operator ω_S .
2. Iterative episode detection. Episodes consisting of an iterated substring are detected and abstracted by applying operator ω_I .
3. Model construction. When necessary, an HMM is constructed for every abstracted episode.
4. Sequence abstraction. The input sequences are rewritten using as new alphabet the names of the abstract episodes.

Non-iterative episode detection. The core mechanism is represented by the abstraction operators ω_S . However, some preprocessing is required before applying the operator. In fact, ω_S takes in input a set S of strings that is constructed by applying a local alignment algorithm LA (see Section 4) followed by a clustering algorithm. More specifically, LA is repeatedly applied to a set of sequence pairs randomly sampled from the learning set \mathcal{LS} and produces in output pairs of subsequences that exhibit strong similarity. It is expected that a frequent episode occurs in many pairs of sequences with minor differences from one instance to another. Then, episodes deriving from a same regular expression are easy to detect by using a clustering algorithm, which groups together most similar subsequences. The currently used algorithm is an incremental variant of classical k-Means. Afterwards, ω_S is applied to every cluster S obtained in this way, constructing a corresponding abstract event.

Iterative episode detection. The procedure described above does not work properly in presence of episodes consisting of iterated substrings. In fact, it is impossible to align two episode instances when the numbers of iterations are different. This problem is solved by operator ω_I , which is applied to a set of sequences sampled from \mathcal{LS} . All iterated episodes found in this way are collected into a set I . Afterwards, episodes characterized by an identical (or very similar) iterated substring are generalized to a unique abstract episode description: a common iterated subsequence is chosen, and the iteration limits are set in order to include all found instances. The abstract events constructed in this way are then added to the ones generated by operator ω_S .

Model Construction. This step is accomplished only if an approximate matching based on HMM has been required and consists in constructing an HMM for every abstract event E characterized in the previous steps. Every expression \mathcal{R}_E is converted into a HMM λ_E , and the sets of substrings, used to learn the regular expression describing the abstract events, are used to estimate the parameters

of λ_E . The details of the algorithm can be found in [2, 11].

Sequence abstraction. Every sequence s in \mathcal{LS} is rewritten into an abstracted sequence s' according to the following algorithm: s is scanned left-to-right searching for instances of episodes detected and abstracted in the previous steps. The presence of an episode E is decided by matching the corresponding regular expression \mathcal{R}_E to s . Every time an instance is found, the *name* of E is appended to s' . However, conflicting interpretations of a same subsequence may exist. Conflict resolution is delayed to a second swept and, initially, a lattice is generated, containing all plausible hypotheses for episode instances. Afterward, lattices are processed extracting from each one the maximum scoring sequence, which includes the best scored hypotheses compatible with the given constraints. The default constraint is that hypotheses must not overlap. In the case a string similarity function of type (2) is used to match regular expressions, the score assigned to episode hypotheses is the value computed by the similarity function. Otherwise, if a matching based on HMM is used, the score of an event E is the probability assigned by the model λ_E . Portions of the string s not abstracted by any episode are abstracted as gaps and represented by a gap symbol.

The major steps of the basic cycle are illustrated through an example in Figure 3.

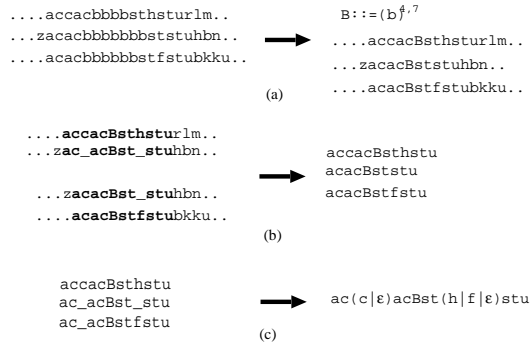


Fig. 3. Basic learning cycle example. (a) Iterated symbols are detected and replaced with the name of the corresponding regular expression. (b) Local alignments are detected and similar substrings are clustered together. (c) From the multiple alignment of elements in a same cluster a regular expression is obtained.

5.4 Refinement cycle

The refinement cycle may be activated at the abstraction layer L_i every time new episodes are detected and modeled at a level higher than i . The reason for doing

it is illustrated in Figure 4. When an episode E is hypothesized and characterized at an abstraction level L_i , the context, i.e, the presence of other episodes before or after E , is not considered. Nevertheless, the context is considered later on when the episodes of layer L_i are linked together into an episode at level L_{i+1} . This means that some instances of E may be not included in any higher level episode and will be considered spurious. Nevertheless, such instances were included in the

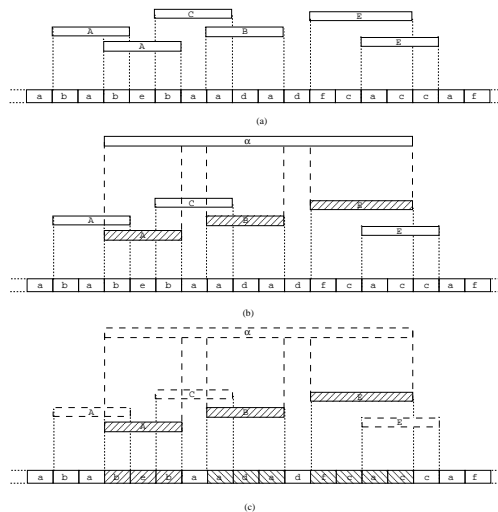


Fig. 4. The refinement step. (a) Episode lattice. (b) Some hypothesized events in (a) are not considered for a new episode. (c) Only the retained instances are used to re-train the episode model.

cluster used to build up the regular expression describing E . In the refinement step, the regular expression describing E are re-learned using only the instances that have been retained.

As episode instances are detected using one of the approximate matching algorithms described in in Section 4, the outcome of the refining cycle heavily depends on it. Therefore, using a similarity function or an HMM can produce quite different results.

6 Discussion and Evaluation

The learning algorithm described in previous sections has been implemented in two different versions, where the major difference is in the flexible matching algorithm. In the first version, which is an extension of the work described in [3], matching is based on a string similarity function. In the second version [2],

string similarity is still used in the basic learning cycle, whereas, in the refinement cycle, regular expressions are translated into HMMs. More specifically, the cascade of regular expressions generated by the abstraction mechanism leads to a Hierarchical HMM [7], which is trained using the classical EM algorithm. The two different versions have been tested both on artificial data and on real world problems. Artificial data are a suitable tool to evaluate learning algorithms, because they can be constructed on purpose to put in evidence both strong and weak points. A first benchmark between the two algorithm versions has been run using a suite of datasets consisting of artificially generated sequences of symbols. The structure for all datasets is similar, and consists of sequences of words selected from natural language and interleaved with gaps filled by randomly chosen characters. Moreover, noise is added by randomly replacing an assigned percentage of characters with other characters randomly extracted from the alphabet. The challenge for the algorithms is to reconstruct the regular expression corresponding to the sequence of words hidden in the data. Every dataset contains 100 sequences and the global length of the sequences ranges from 60 to 140 characters.

The difficulty of the task has been controlled by varying three parameters: (a) the number of words ($5 \leq w \leq 8$) in the regular expression; (b) the word length ($5 \leq L \leq 8$); (c) the noise level ($N \in \{0\%, 5\%, 10\%, 15\%\}$). For every triple $\langle w, L, N \rangle$, 10 different datasets have been generated for a total of 640 learning problems.

The results comparing the two algorithms are summarized in Table 1. The error rate (averaged on 10 problems) is evaluated as the edit distance (i.e., the minimum number of corrections) between the regular expression learned by the algorithms and the original one hidden in the data. When an entire word is missed, the corresponding error is set equal to its length. Experiments in Table 1, reporting an error rate much higher than the others, have missed words. In all cases, the learning cycle has been iterated twice, as explained in Section 5. Cells in the last row reports the average of the error rates in the corresponding column.

From the results of Table 1 the two algorithm versions show performances substantially similar, and show good robustness with respect to the presence of noise. However, from a more detailed analysis of the table, some differences emerge. Flexible matching based on HMM is slightly superior to similarity based one when the word length increases. Moreover, it shows a less stable behavior: either the performances are very good, with a zero error rate, or the error rate is quite high, because one or more words have been missed in some learning problem.

It is worth noting that both versions exhibit better performances when episodes to detect are longer. This is easy to explain, because long-range regularities are easier to distinguish from noise than short-range ones. Then, in general, longer CEs are expected to be easier to learn than shorter ones.

As previously mentioned, the learning algorithm has been applied also to a real world problem where the goal was to learn the profile of a user typing on a

Table 1. Performances of the two algorithm versions obtained on artificial datasets. The sequence length ranges from 60 to 140 characters. The CPU time for solving a problem ranges from 42 to 83 seconds on a Pentium IV 2.4 Ghz.

		Similarity function				HMM			
w	L	Noise Level				Noise Level			
		0%	5%	10 %	15%	0%	5%	10 %	15%
5	5	0.00	0.00	0.00	0.00	0.04	0.04	0.04	0.04
5	6	0.01	0.01	0.02	0.01	0.03	0.03	0.03	0.03
5	7	0.00	0.03	0.02	0.02	0.00	0.00	0.02	0.00
5	8	0.01	0.03	0.03	0.04	0.00	0.00	0.00	0.00
6	5	0.00	0.02	0.03	0.02	0.10	0.06	0.00	0.03
6	6	0.02	0.04	0.01	0.01	0.05	0.00	0.00	0.00
6	7	0.03	0.02	0.02	0.04	0.02	0.00	0.00	0.00
6	8	0.05	0.06	0.04	0.04	0.00	0.00	0.04	0.00
7	5	0.02	0.01	0.01	0.01	0.02	0.05	0.01	0.10
7	6	0.02	0.00	0.01	0.01	0.04	0.02	0.05	0.04
7	7	0.04	0.02	0.02	0.04	0.00	0.00	0.02	0.05
7	8	0.04	0.02	0.02	0.04	0.01	0.00	0.09	0.09
8	5	0.03	0.01	0.01	0.01	0.00	0.00	0.01	0.00
8	6	0.01	0.00	0.01	0.01	0.03	0.06	0.06	0.14
8	7	0.07	0.03	0.04	0.05	0.00	0.00	0.00	0.00
8	8	0.12	0.06	0.05	0.07	0.01	0.00	0.00	0.00
Avg.		0.026	0.022	0.021	0.026	0.021	0.016	0.023	0.026

keyboard. The description of the task and the obtained results are described in [11]. In this case, the algorithm version based on HMM clearly dominated the other version. The reasons are to be searched in the greater complexity of the data set and on the nature of the data, which better fit an HMM. In fact, the similarity function (2) does not take into account the location of the errors in the episodes whereas HMM does. Therefore, it is not possible to establish a-priori which version would perform better, if the nature of the data is not known.

7 Conclusion

An algorithm for discovering complex events in noisy sequences has been presented, where complex events are described as regular expressions. The main contribution of the paper consists in organizing and generalizing in a unique framework different methods developed in the past. Moreover, for the first time the learning algorithm architecture, which is based on an abstraction mechanism, is described into details.

Currently, two versions of the algorithm exist, which have been developed by integrating large part of previous work: one makes use of string similarity in order to match regular expressions on noisy data; the other one translates regular expressions into a hierarchical Hidden Markov Model [7]. Both versions exhibit good performances on artificial data, whereas the second one was superior in

solving a non trivial user profiling problem, where it was required to learn the model for a user editing a text.

In both versions, the algorithm has been easy to apply and didn't require special tuning on the problem. This means that the method is robust and suitable for applications in real domains.

However, the evaluation is not yet complete. On the one hand, the ability of the algorithm at learning regular expressions where iteration is deeply involved has been only partially tested, and the results obtained up to now are not yet conclusive. On the other hand, a theoretical analysis of the convergency properties of the algorithm is in progress.

Acknowledgments

The present work has been supported by the FIRB Project: WebMinds.

References

1. D. Angluin. Queries and concept learning. *Machine Learning*, 2(4):319-342, 1988.
2. M. Botta, U. Galassi, and A. Giordana. Learning complex and sparse events in long sequences. In *Proceedings of the European Conference on Artificial Intelligence, ECAI-04*, Valencia, Spain, August 2004.
3. M. Botta, A. Giordana, and P. Terenziani. Discovering complex events in long sequences. In *Proceedings of the "Workshop on learning in temporal sequences", Machine Learning Conference*, Sidney, Australia, July 2002.
4. F. Denis. Learning regular languages from simple positive examples. *Machine Learning*, 44(1/2):37-66, 2001.
5. R. Durbin, S. Eddy, A. Krogh, and G. Mitchison. *Biological sequence analysis*. Cambridge University Press, 1998.
6. J. L. Elman. Distributed representations, simple recurrent networks, and grammatical structure. *Machine Learning*, 7:195-225, 1991.
7. S. Fine, Y. Singer, and N. Tishby. The hierarchical hidden markov model: Analysis and applications. *Machine Learning*, 32:41-62, 1998.
8. P. Frasconi and Y. Bengio. An em approach to grammatical inference: iputo/output hmms. In *Proceedings of International Conference on Pattern Recognition, ICPR-94*, 1994.
9. K. S. Fu. *Syntactic pattern recognition and applications*. Prentice Hall, 1982.
10. K.S. FU and T.L. Booth. Grammatical inference: Introduction and survey (part 1). *IEEE Transaction on System, Men and Cybernetics*, 5:85-111, 1975.
11. U. Galassi, A. Giordana, and D. Mendola. Learning user profiles from traces. *Technical report TR-INF-2005-04-02-UNIPMN*, 2005.
12. D. Gussfield. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, 1997.
13. J.E. Hopcroft and J.D. Ullman. *Formal languages and their relation to automata*. Addison-Wesley, 1969.
14. K. Murphy and M. Paskin. Linear time inference in hierarchical hmms. In *Advances in Neural Information Processing Systems (NIPS-01)*, volume 14, 2001.
15. E.W. Myers and W. Miller. Approximate matching of regular expressions. *Bulletin of Mathematical Biology*, 51(2):5 - 37, 1989.

16. R. G. Parekh and V. G. Honavar. Learning DFA from simple examples. In *Proceedings of the 8th International Workshop on Algorithmic Learning Theory (ALT'97), Lecture Notes in Artificial Intelligence*, volume 1316, pages 116–131, Sendai, Japan, 1997. Springer.
17. Rajesh Parekh, Codrin Nichitiu, and Vasant Honavar. A polynomial time incremental algorithm for learning DFA. *Lecture Notes in Computer Science*, 1433:37–50, 1998.
18. S. Porat and J. Feldman. Learning automata from ordered examples. *Machine Learning*, 7:109–138, 1991.
19. P. Garcia and E. Vidal. Inference of k-testable languages in the strict sense and applications to syntactic pattern recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(9):920–925, 1990.
20. L. Saitta, editor. *The abstraction paths, Special issue of the Philosophical Transactions of Royal Society, Series B*. 2003.
21. M. Skounakis, M. Craven, and S. Ray. Hierarchical hidden markov models for information extraction. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence IJCAI-03*, pages x–x. Morgan Kaufmann, 2003.
22. L. Xie, S. Chang, A. Divakaran, and H. Sun. *Learning hierarchical hidden Markov models for video structure discovery*, volume Tech. Rep. 2002-006. ADVENT Group, Columbia University, December 2002.