

Applying a Finite Automata Acquisition Algorithm to Named Entity Recognition

Muntsa Padró and Lluís Padró

TALP Research Center
Universitat Politècnica de Catalunya
{mpadro,padro}@lsi.upc.edu

Abstract. In this work, Causal-State Splitting Reconstruction algorithm, originally conceived to model stationary processes by learning finite state automata from data sequences, is for the first time applied to NLP tasks, namely Named Entity Recognition. The obtained results are slightly below the best systems presented in CoNLL 2002 shared task, though given the simplicity of the used features, they are really promising.

Once the viability of using this algorithm for NLP tasks is stated, we plan to improve the results obtained at NER task, as well as to apply it to other NLP sequence recognition tasks such as PoS tagging, chunking, subcategorization patterns acquisition, etc.

1 Introduction

Some Natural Language Processing (NLP) tasks may be naturally approached using finite state automata and machine learning algorithms. These automata can be hand built with linguistic knowledge or can be statistical models, such as Hidden Markov Models (HMM). In the case of statistical automata, usually their structure must be previously defined. For HMM, for example, it is necessary to define what the states represent, and the statistics are only applied to learn the transition and emission probabilities. Nevertheless there are algorithms that learn automata given some data [1–6]. One of these kind of algorithms is CSSR (Causal State Splitting Reconstruction) which is based on inferring the causal states of a process given sequential data.

In this work a first approach to applying this algorithm to NLP tasks is presented. The task chosen to start applying this algorithm was Named Entity Recognition (NER). The results presented in this paper are preliminary, since the performed experiments take into account few features. Nevertheless, the obtained results are quite promising since they are not far from those of the state of the art systems and there is still a large margin for improvement to the presented preliminary experiments. At the sight of current results, it can be said that this algorithm can be reliably applied to NER and we expect to obtain good results in the future applying it to other NLP tasks.

The Named Entity Recognition (NER) task consists of detecting names referring to entities such as persons, locations, organizations, etc. in a text. This

is used in many NLP applications such as Question Answering, Information Retrieval, Summarization, etc. Furthermore, some Named Entity Classification (NEC) systems need to have the Named Entities (NE) previously detected. Other systems perform both tasks (detection and classification) at the same time. We only applied CSSR to the detection step, not to classification.

The rest of the paper is organized as follows: section 2 presents the CSSR algorithm and its theoretical basis. Section 3 defines our approach to apply the algorithm to NER task. In section 4 the performed experiments with the obtained results are discussed. Section 5 states some conclusions and future work.

2 CSSR algorithm

The CSSR algorithm [7] performs the blind construction of asymptotically optimal nonlinear predictors of discrete sequences. It infers the causal states from data, searching for optimal predictors for discrete random processes, in the form of Markov Models.

2.1 Causal States

Given a discrete alphabet Σ , consider a sequence x^- drawn from Σ (history) and a random variable for future sequences Z^+ . Z^+ can be observed after x^- with a probability $P(Z^+|x^-)$. Two histories, x^- and y^- , are equivalent when $P(Z^+|x^-) = P(Z^+|y^-)$, i.e. when they have the same probability distribution for the future.

The different future distributions build the equivalence classes which are named *causal states* of the process. Each causal state is a set of history suffixes, up to a preestablished maximum length, with the same probability distribution for the future. The causal states of a process form a deterministic machine and are recursively calculable.

2.2 The algorithm

Causal-State Splitting Reconstruction (CSSR) estimates an HMM inferring the causal states from sequence data. The main parameter of this algorithm is the maximum length (l_{max}) the suffixes can reach. That is, the maximum length of the considered histories. In terms of HMMs, l_{max} would be the potential maximum order of the model (the HMM would have l_{max} order if all the suffixes belonged to different states).

The algorithm starts by assuming the process is an identically-distributed and independent sequence with a single causal state, and then iteratively adds new states when it is shown by statistical tests that the current states set is not sufficient. The causal state machine is built in three phases (see [7] for details):

1. Initialize:

Create a state set with only one state containing only the null suffix. Set $l = 0$ (length of the longest suffix so far).

2. Sufficiency:

Iteratively build new states depending on the future probability distribution of each possible suffix extension (suffix sons). Before doing so it is necessary to estimate the probability distribution $\hat{P}(X_t|S=s)$ (where X_t is the random variable for the next alphabet symbol in the sequence) for each state s . This is necessary because this probability can change at each iteration when the new suffixes are added to a given state. This probability distribution is estimated (via maximum likelihood, for instance) using the data.

At this phase, the suffix sons (ax) for each longest suffix (x) are created adding each alphabet symbol (a) at the beginning of each suffix. The future distribution $P(X_t|X_{t-l}^{t-1})$ (probability of each alphabet symbol given the last l symbols) for each son is computed and a hypothesis test with the following null hypothesis is performed,

$$P(X_t|X_{t-l}^{t-1}=ax)=P(X_t|S=s); \quad \forall a \in \Sigma$$

This hypothesis is true if the new distribution is equal (with a certain confidence degree) to the distribution of an existing state (s). In this case, the suffix son is added to this state. If the hypothesis is rejected for all states, a new state for the suffix son is created. To check the null hypothesis we can use a statistical test such as χ^2 or Kolmogorov-Smirnov.

As the suffix length grows, l is increased by one at each iteration. This phase goes on until l reaches some fixed maximum value l_{max} , the maximum length to be considered for a suffix, which represents the longest histories taken into account. The results of the system will be significantly different depending on the chosen l_{max} value, since the larger this value is, the more training data will be necessary to learn a correct automaton with statistical reliability. Also, the time needed to learn the automaton grows linearly with l_{max} . So it is necessary to tune the best maximum length for the amount of available data (or viceversa, the amount of necessary data for the required suffix length).

3. Recursion:

Since CSSR models stationary processes, first of all the transient states are removed. Then the states are splitted until a deterministic machine is reached. To do so, the transitions for each suffix in each state are computed and if two suffixes in one state have different transitions for the same symbol, they are splitted into two different states.

At the end of this recursion phase, a deterministic automaton is obtained.

In figure 4 the pseudo code for this algorithm is presented. See [7] for extended details and algorithm analysis.

3 Applying CSSR to Named Entity Recognition

In this work an approach to apply CSSR algorithm to Named Entity Recognition is presented. We only worked on recognizing Named Entities, not in classifying them.

Following CoNLL 2002 and 2003 shared task, we worked with the “B-I-O” approach [8]. Each word has a B, I or O tag, being B the tag for a word where a NE begins, I the tag if the word is part of a NE but not the beginning, and O the tag for the words not belonging to any NE. There are other possible approaches to tagging NEs [9] but this is one of the most widely used.

The general idea of our approach is to use CSSR to learn an automaton for NE structure. Once the automaton is learnt, it can be applied to detect NEs in untagged text.

3.1 Learning the automaton

To learn the automaton that must reproduce NE structure, different information about the words is used. This information can be orthographic, morpho-syntactic, about the position in the sentence, etc. Using this features, the words in a sentence are translated to a closed set of symbols, that will be the alphabet of the automaton. The sentence translated in such a way will be the sequence that we use to learn the automaton via CSSR.

A problem of using that algorithm for this task is that it is conceived to model stationary processes, but NE patterns are not in this category. So, what we did was to regard a text sequence as a stationary process in which NEs occur once a while. Doing so implies the automaton is modelling the pattern of the sequence (the text), not the pattern of a NE.

To allow CSSR to learn the pattern of the NEs, we introduce in the alphabet the information of the NE-tag (B, I or O) available in the supervised training corpus. So the correct NE-tag is taken into account for each kind of word when building the automaton.

To allow CSSR to learn the pattern of the NEs, we introduce in the alphabet the information of the NE-tag (B, I or O) available in the supervised training corpus. Thus, the hidden information (the tags) is taken into account when building the automaton.

In this way, although we obtain an automaton modelling the entire text sequence as an stationary process, we have information encoded in the transitions about B-I-O tags for NEs in the text. Thus, we can later use this information to compute the best path for a sequence and use it to tag NEs in a new text.

3.2 An Example

For instance, let’s suppose an approach where the only feature taken into account is whether a word is capitalized or not. Let’s say that a capitalized word will have the feature “A” and a non-capitalized word the feature “a”. In this case, the alphabet will consist of six symbols, which are the possible combinations of a capitalization value and a B-I-O tag (A_B , A_I , A_O , a_B , a_I , a_O). Each word will be translated into sequences of these symbols depending on whether it is capitalized and on its NE-tag.

Figure 1 shows an example of a possible training sentence and its translation to this alphabet. The first two columns would be the sentences as they are in the

training corpus: a word and its right B-I-O tag. The last column is their translation into the alphabet, which will be used as input for the CSSR algorithm. In this example it becomes clear that this alphabet would be too poor to capture appropriate NE patterns. It would be necessary, for example, to introduce information about the beginning of sentences (where all words appear capitalized and may not be a NE), or to introduce special words that may appear uncapitalized inside a NE (prepositions, articles...).

Word	Correct Tag	Alphabet Symbol
Yesterday	O	A_O
the	O	a_O
President	B	A_B
of	I	a_I
France	I	A_I
spoke	O	a_O
with	O	a_O
George	B	A_B
Bush	I	A_I
about	O	a_O
the	O	a_O
situation	O	a_O
in	O	a_O
Iraq	B	A_B
.	O	a_O
Bush	B	A_B
said	O	a_O
...		

Fig. 1. Example of a training sentence and its translation to a simple alphabet

Once the data are properly translated into the alphabet, the automaton is built using CSSR. Figure 2 shows a possible (not real) automaton learned with CSSR with the alphabet $\{ A_B, A_I, A_O, a_B, a_I, a_O \}$.

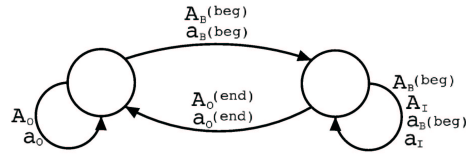


Fig. 2. Example of an automaton that models simple NEs

3.3 Using the learned automaton to tag NEs

When a sentence has to be tagged, the information about the correct NE tag is not available, so there are several possible alphabet symbols for that word. It is only possible to know the part of the translation that depends on the word or sentence features. In our example, it would be possible to translate each word to an “A” or to an “a”, but not to know the part of the symbol that depends on the NE-tag, which is, in fact, what we want to know.

To find this most likely tag for each word in a sentence –that is, to find the most likely symbol of the alphabet (e.g. G_B , G_I , G_O for a G word), a Viterbi algorithm is applied. That is, for each word in a sentence, the possible states the automaton could reach if the current word had the tag B, I, or O, and the probabilities of these paths are computed. Then, only the highest probability for each tag is recorded. That means that for each word, the best path for this word having each tag is stored. At the end of the sentence, the best probability is chosen and the optimal path is backwards recovered. In this way, the most likely sequence of B-I-O tags for each word in the sentence is obtained. There are some forbidden paths, which are those that lead to the OI tag-combination. The paths including this combination are pruned out.

3.4 Managing Unseen Transitions

When performing the tagging of NEs given a text, it is possible to find symbol sequences that haven’t been seen in the training corpus. This will cause the automaton to fall in a *sink* state, which receives all the unseen transitions. This state can be seen as the state that contains all the unseen suffixes. All unseen transitions probabilities are smoothed to have a small probability of arriving to the sink state. Actually, the only sequences that have zero probability are those that have a forbidden combination of tags or of states being the beginning or the end of a NE.

When the automaton falls in the *sink* state, it can not follow the input sequence using transition information because, as the transitions weren’t seen, they are not defined. To allow the system to continue tagging the text, when the automaton falls into *sink* state, the suffix of length l_{max} is built using the last $l_{max} - 1$ symbols and the next symbol from the input. A state containing this new suffix is searched over the automaton and, if found, the automaton goes to this state and continues its normal functioning. If not, the process is repeated, getting more symbols from the input sequence, until a state containing the new suffix is found.

This may cause skipping some part of the input, and is caused by the fact that the text sequence is considered as an stationary process, and so, when the CSSR-acquired automaton fails, we have to resynchronize it with the input data.

4 Experiments and Results

In this work, the data for the CoNLL-2002 shared task [10] for Spanish were used. These data contain three corpora: one for the train and two for the test:

one for the development of the system and the other one for the final test. The amount of data in each corpus is shown in table 1.

Corpus	Number of words	Number of NEs
Train	264,715	18,797
Test a	52,923	4,351
Test b	51,533	3,558

Table 1. Number of words and NEs in each corpus

With these data, two different kind of experiments were performed: one to validate the method, and the other one to evaluate it over real data.

4.1 Validating the Method

The experiments to validate the method consist of tagging the training and test corpora using a simple hand-built automaton, and checking whether CSSR is able to learn and reproduce its behaviour. FreeLing analyzer [11] has a NE system that uses a simple hand-built automaton of four states, and was used to re-annotate all CoNLL datasets, obtaining training and test corpora tagged with a simple and systematic annotation criteria. These corpora were used to train and test CSSR at the NE task.

The used alphabet was the same –and encoded the same features– than the one used by FreeLing NE detection module, in which the following feature-sets are mapped to the alphabet symbols:

- **G**: Beginning of the sentence, capitalized, not containing numbers, not in the dictionary¹.
- **S**: Beginning of the sentence, capitalized, not containing numbers, one of its possible analysis being a common noun.
- **M**: Not at the beginning of the sentence, capitalized.
- **a**: Not at the beginning of the sentence, non-capitalized, functional word ².
- **w**: Other.

In this way, the alphabet for CSSR will be the combination of these four features with the three possible NE tags (G_B , G_I , G_O , S_B , S_I , S_O , M_B , M_I , etc.).

Using the training corpus tagged with FreeLing an automaton was learned using CSSR algorithm. Then, this automaton was evaluated over the test corpora (also tagged with FreeLing). The system obtained $F_1 = 100\%$ when using $l_{max} = 2$ for both test sets, and using $l_{max} = 3$, $F_1 = 99,83\%$ for the development corpus (test a) and $F_1 = 99,98\%$ for the test corpus (test b) were obtained. For

¹ The used dictionary is the one provided by FreeLing [11].

² Functional words are articles or preposition that are often found inside a NE.

$l_{max} = 3$, the lost in F_1 is due to some missed NEs. In fact the system obtained a 100% precision in both cases, but the recall fell a little bit. This loss in the recall, could be due to the fact that if l_{max} rises, more training data are necessary to generate a correct automaton.

These results prove that CSSR is able to perfectly acquire the behaviour of the FreeLing annotation schema underlying the data. Although this is an easy task, since that schema is simple and systematic, it validates the viability of our adaptation of CSSR from stationary-process acquisition to pattern recognition.

4.2 Applying CSSR over the real corpora

Next step was testing the method on a real NE task, over the corpora used in CoNLL-2002. In this case, the annotation was hand made, and thus, it will include cases much more complex than the naive FreeLing annotation, and also present noise and inconsistencies due to different annotator criteria or simply to human mistakes.

CSSR algorithm has three important parameters. One is the chosen maximum length (l_{max}), which is the most significant parameter. The other two are the test used to check the null hypothesis and the parameter α , controlling the test significance degree. We made several experiments (using the same alphabet presented in 4.1) for different l_{max} values and with two different statistical tests: χ^2 and Kolmogorov-Smirnov. For each test, the experiments were performed with several α values. Figure 3 shows the obtained results with the different automata built using Kolmogorov-Smirnov test. The results obtained with χ^2 test have a similar behaviour but are slightly worse.

In this figure it can be seen that the significance degree value is not as influential as the l_{max} value. In fact, there is a range of α values for which the reached results are similar.

About the influence of l_{max} , the results show that best performance is obtained with small l_{max} , likely caused by the limited size of the training corpus, which seems not to allow statistically reliable acquisition of automata with too long histories. In fact, since our alphabet has 16 symbols, the number of suffixes of length 5 is 16^5 (over one million), which is approximately the size of the training corpus, so most of the possible suffixes wouldn't have been seen in the corpus.

The best performance is obtained with $l_{max} = 3$ and $\alpha = 1e - 5$. With these values, the system reaches a precision of 89.81%, a recall of 88.22% and $F_1 = 89.01\%$ for the development corpus (test a) and a 90.03% precision, 88.81% recall and $F_1 = 89.42\%$ for the test corpus (test b).

These results can be compared with the winner system of CoNLL-2002 shared task [12]. This system was developed with the same training and testing data and performs the NE recognition and classification separately, so it is possible to compare our system with the part that performs the NE recognition.

That system obtained a F_1 of 91.66% for the Spanish development corpus and a 92.91% for the test corpus. These results are higher than the results presented in this work, which was expected since the feature set used by that system is

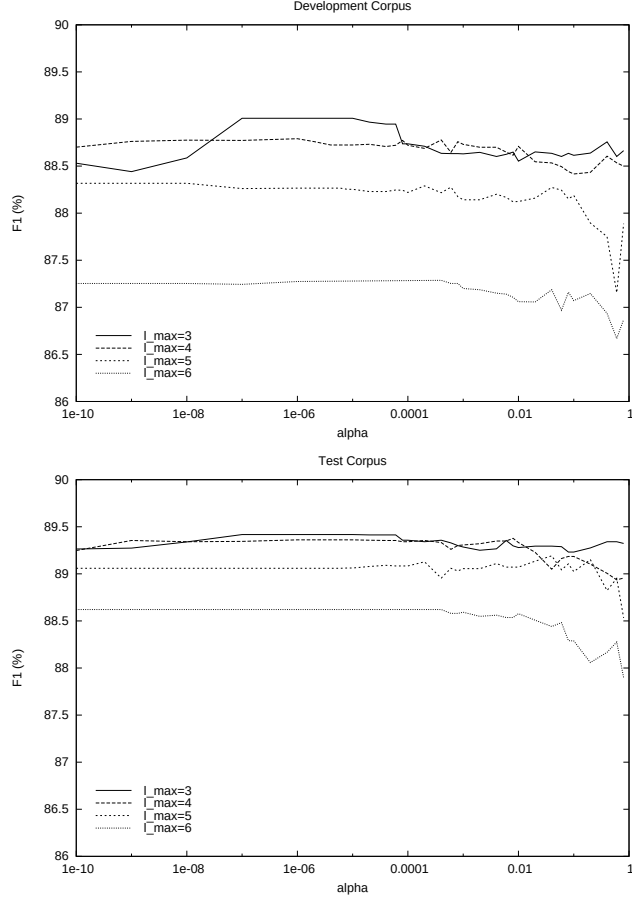


Fig. 3. Obtained results with different l_{max} and α values for both test corpora

much richer (bag of words, disambiguated PoS tag, many orthographic features, etc.) than the used in our experiments.

Furthermore, it is possible to apply the NEC system used by [12] to the output of our NE detector. Doing so over our best results yields to a $F_1 = 76.30\%$, which would situate our system in the fifth position in CoNLL-2002 ranking table for complete NER systems in Spanish.

Another factor that is interesting to study is the number of generated states for each configuration. As it is expected, l_{max} and α values not only affect the performance of the system, but also change the number of states of the generated automata. The larger l_{max} and α are, the greater the number of states will be. As in the case of the system performance, the most influent parameter in the automata size is l_{max} , while the influence of α is important only for values over 0.01. Using values under this threshold, the number of generated states varies

from 100 states for $l_{max} = 3$ to 2000 states for $l_{max} = 6$. For α bigger than 0.01 the number of states rises until these values are duplicated.

5 Conclusions and Further Work

In this work a finite automata acquisition algorithm has been applied to Named Entity Recognition. The algorithm learns automata for stationary processes, so, some arrangements have had to be done in the tagging step to fit a non-stationary pattern recognition NLP task such as NE recognition.

Firstly, the method has been validated by applying it to learn sentence patterns from a corpus annotated with a simple hand-made automaton, and checking that CSSR is able to exactly reproduce its behaviour.

Secondly, it has been shown that this algorithm can build automata that give pretty good results when applied to recognize the NEs of a text. In fact, the system results are not too far from those obtained by the winner system on CoNLL 2002 shared task and they may be expected to improve by introducing more information in the system, since we use a much simpler knowledge than all CoNLL 2002 participants.

The main conclusion of this work, is that CSSR algorithm can be satisfactorily applied to NER tasks, which opens a door to applying it to other basic NLP tasks which need to learn sequential pattern information from data (PoS tagging, chunking, etc.)

The future work to be developed is focused on improving this NER system and on applying CSSR algorithm to other NLP tasks. To improve NER, more orthographic and morpho-syntactic information will be introduced in the alphabet in order to build more accurate automata. Similarly, external information such as trigger word lists or gazetteers could be also used. Other NLP tasks where this algorithm can be applied are chunking, PoS tagging or subcategorization pattern acquisition.

Acknowledgements

This research is being funded by the Catalan Government Research Department (DURSI), by the Spanish Ministry of Science and Technology (ALIADO TIC2002-04447-C02) and by the European Commission projects: Meaning (IST-2001-34460) and CHIL (IST-2004-506909). Our research group, TALP Research Center, is recognized as a Quality Research Group (2001 SGR 00254) by DURSI.

References

1. Segarra, E., Sanchis, E., García, F., Hurtado, L.F., Galiano, I.: Achieving full coverage of automatically learnt finite-state language models. In: Workshop on Finite-State Methods in Natural Language Processing. 10th Conference of the European Chapter of the Association for Computational Linguistics (EACL2003), Budapest, Hungary (2003) 135–142

2. Pla, F.: Etiquetado Léxico y Análisis Sintáctico Superficial basado en Modelos Estadísticos. PhD thesis, Departament de Sistemes Informàtics i Computació, Universitat Politècnica de València (2000)
3. Lang, K.J.: Random dfa's can be approximately learned from sparse uniform examples. In: COLT '92: Proceedings of the fifth annual workshop on Computational learning theory, ACM Press (1992) 45–52
4. Oncina, J.M.: Aprendizaje de lenguajes regulares y funciones subsecuenciales. PhD thesis, Departamento de Sistemas Informáticos y Computación, Universidad Politécnica de Valencia (1991)
5. Rulot, H., Vidal, E.: Modelling (sub)string-length based constraints through a grammatical inference method. In: Pattern recognition theory and applications. Springer-Verlag (1987) 451–459
6. Trakhtenbrot, B., Barzdin, Y.: Finite Automata: Behaviour and Synthesis. North Holland Publishing Company (1973)
7. Shalizi, C., Shalizi, K.: Blind construction of optimal nonlinear recursive predictors for discrete sequences. In: Uncertainty in Artificial Intelligence: Proceedings of the Twentieth Conference. (2004)
8. Ramshaw, L., Marcus, M.P.: Text chunking using transformation-based learning. In: Proceedings of the Third ACL Workshop on Very Large Corpora. (1995)
9. Tjong Kim Sang, E.F., Veenstra, J.: Representing text chunks. In: Proceedings of EACL'99, Bergen, Norway (1999) 173–179
10. Tjong Kim Sang, E.F.: Introduction to the conll-2002 shared task: Language-independent named entity recognition. In: Proceedings of CoNLL-2002, Taipei, Taiwan (2002) 155–158
11. Carreras, X., Chao, I., Padró, L., Padró, M.: Freeling: An open-source suite of language analyzers. In: Proceedings of the 4th International Conference on Language Resources and Evaluation (LREC'04), Lisbon, Portugal (2004)
12. Carreras, X., Màrquez, L., Padró, L.: Named entity extraction using adaboost. In: Proceedings of CoNLL Shared Task, Taipei, Taiwan (2002) 167–170

Algorithm CSSR ($\Sigma, \bar{x}, l_{max}, \alpha$)

```

 $l \leftarrow 0$ 
 $q_0 \leftarrow \{\lambda\}; Q \leftarrow q_0$ 

while  $l < l_{max}$ 
  for each  $s \in Q$ 
    estimate  $\hat{P}(X_t | S = s)$ 
    for each  $x \in s$ 
      for each  $a \in \Sigma$ 
        estimate  $p \leftarrow \hat{P}(X_t | X_{t-l}^{t-1} = ax)$ 
        Reorganize_States( $Q, p, ax, s, \alpha$ )
   $l \leftarrow l + 1$ 

Remove transient states from  $Q$ 
repeat
   $recursive \leftarrow True$ 
  for each  $s \in Q$ 
    for each  $b \in \Sigma$ 
       $x_0 \leftarrow \text{first } x \in s$ 
       $T[s, b] \leftarrow \text{Class}(x_0 b, Q)$ 
      for each  $x \in s, x \neq x_0$ 
        if  $\text{Class}(xb, Q) \neq T[s, b]$ 
          create new state  $s' \in Q$ 
           $T[s', b] \leftarrow \text{Class}(xb, Q)$ 
          for each  $y \in s | \text{Class}(yb, Q) = \text{Class}(xb, Q)$ 
            Move_Suffix( $y, s, s'$ )
           $recursive \leftarrow False$ 
until  $recursive$ 

function Reorganize_States( $Q, p, y, s, \alpha$ )
  if null hypothesis passes a test of size  $\alpha$  for  $s$ 
     $s \leftarrow y \cup s$ 
  else
    if null hypothesis passes a test of size  $\alpha$ 
      for  $s^* \in Q, s^* \neq s$ 
         $s' = s^*$ 
      else
         $Q \leftarrow s'$ 
      Add_Suffix( $y, s'$ )

function Add_Suffix( $y, s$ )
   $s \leftarrow s \cup y$ 
  re-estimate  $\hat{P}(X_t | \hat{S} = s)$ 

function Class( $y, Q$ )
  return ( $s \in Q | y \in s$ )

function Move_Suffix( $y, s_1, s_2$ )
   $s_1 \leftarrow s_1 \setminus y$ 
  re-estimate  $\hat{P}(X_t | \hat{S} = s_1)$ 
   $s_2 \leftarrow s_2 \cup y$ 
  re-estimate  $\hat{P}(X_t | \hat{S} = s_2)$ 

// Initialization: initialize the machine with a state with
// only the null suffix.

// Sufficiency: build causal states.

// Estimate the probability distribution for next symbol given the
// current state since it may have changed in the last iteration.

// Estimate the probability distribution for this
// suffix son and perform the hypothesis test.

// Recursion: remove transient states, makes the machine
// deterministic and fills transition table ( $T[\text{state}, \text{symbol}]$ )

// Look for the transition for this suffix with this symbol.

// If the transition for another suffix goes to a different
// state, creates a new state with this suffix and move
// to this state all the suffix with the same transition.

// Reorganize_States: test the null hypothesis and decide to
// which state a suffix must be added or create a new state.
// If the probability distribution for  $y$  is equal to
// the  $s$  distribution add  $y$  to this state.

// If the probability distribution for  $y$  is equal to the  $s^* \neq s$ 
// distribution, add  $y$  to this state  $s^*$ .

// If the probability distribution for  $y$  is different from that
// of all states, create a new state and add  $y$  to it.

// Add_Suffix: add a suffix to a state.

// Class: return the causal state (equivalence class)
// a suffix belongs to.

// Move_Suffix: move a suffix from one state to another.

```

Fig. 4. Pseudo code for the CSSR algorithm.