Refining learning models in grammatical inference

Wang, Xiangrui

2008

Wang, X. (2008). Refining learning models in grammatical inference. Doctoral thesis, Nanyang Technological University, Singapore.

https://hdl.handle.net/10356/13589

https://doi.org/10.32657/10356/13589

NANYANG TECHNOLOGICAL UNIVERSITY

Refining Learning Models in Grammatical Inference

A Thesis Submitted to the School of Computer Engineering
Of the Nanyang Technological University, Singapore

by

Xiangrui Wang

In Partial Fulfillment of the Requirements for the
Degree of Doctor of Philosophy

Mar 17, 2008

Acknowledgements

I would like to express my warm thanks to Dr. Narendra S. Chaudhari for understanding and encouragement during my research work, for his kindly supervision in every aspect of Ph.D. work, and for detailed comments of earlier versions of this thesis.

I may express my gratitude to our wonderful colleagues, who have given advices, feedback and support about my research work. Suffice it to say that to Ms. Xu Yi, Ms. Chen Jinmiao, Mr. Zhu Lin, Mr. Chen Fei, Ms. Ma Yang, Mr. Ho Nhu Binh and Ms. Wang Di, Thank you! Thanks Mr. Lau Boon Chee and Mr. Tan Swee Huat for their kindly support in center of computational intelligence.

I also would like to express my appreciation for all the efforts of everyone helped me in my research work : Dr. Cai Wentong, Dr. Sourav, Dr. Ng Wee Keong, Dr. Wlodzislaw Duch in NTU, Dr. Bill Smyth in McMaster University, Mr. Hamid Abdul Basit and Dr. Stanislaw in NUS.

I am indebted always to my parents and my wife with their untiring support and unlimited belief in me, thanks.

Table of Content

List of Figures

List of Tables

Glossary

Σ	Alphabet
λ	Empty String
ABL	Alignment Based Learning
CFG	Context Free Grammar
CFL	Context Free Language
DFA	Deterministic Finite Automata
DFCA	Deterministic Finite Cover Automata
I	Initial state set of a deterministic acceptor
I^t	Input node in recurrent neural network
PAC	Probably Approximately Correct
PBAL	Profile Based Alignment Learning
RNN	Recurrent Neural Network
SCFG	Stochastic Context Free Grammar

Abstract

Grammatical inference is a branch of computational learning theory that attacks the problem of learning grammatical models from string samples. In other words, grammatical inference tries to identify the computational models that generate the sample strings. In recent decade, due to the explosion of information, efficient ways of searching and organizing are of great importance. Therefore, using grammatical inference methods to process information computationally is very interesting. In real applications, the space-cost plays an important role on performance. In this thesis, we address the problem of refining learning models in grammatical inference.

For regular language learning, we introduce formally the notion of "Classification Automaton" that reduces model size by identifying one automaton for multiple string classes. Classification automaton is proved to reduce 30% model size from a straightforward multiple automata approach on house rent data obtained from the public folder in Microsoft Exchange Server of Nanyang Technological University. In real world applications, there is always a maximum possible length for the strings. Based on this observation, we further introduce cover automata, which simplified a learning model with a maximum length limit, for grammatical inference. Test results based on Splice-junction Gene Sequence database demonstrate the method reduces model size by 32% from the widely used deterministic finite automaton model. By mixed k-th order Markov Chains, stochastic information for all possible substrings within k+1 length is captured. However, the space cost is exponential. We introduce the use of recurrent neural networks (RNNs) and present a pruning learning method to avoid the exponential space costs. There is a tradeoff between the accuracy and

model size. We proposed a balancing method and presented test results based on Splice-junction Gene Sequence database, which demonstrate that the method reduced the model size by 105 times with a reduced accuracy from 80% to 76%. Then, we introduce profile-based alignment learning (PBAL) framework to refine the model for context-free grammar learning. The PBAL framework is an extension of the existing Alignment Based Learning (ABL) framework by making use of statistical information to refine alignment and further refine the learned grammatical rules. The converged results rules are proved in experiments to improve the alignment precision from 50% to 90% with model size reduced to 47%.

CHAPTER

1

Introduction

Computational learning theory is a branch of computer engineering that studies how to design computer programs that are capable of exhibiting learning; it further identifies the computational limit of learning by machines. The learning models study abstractions from real life problems. Thus close connections with experimentalists are useful to validate or modify these abstractions so that the results could help to explain or predict empirical performance. One branch of the computational learning theory, called *Grammatical Inference* (GI), attacks the problem of learning grammatical models from samples. The applications of grammatical inference can be found in many fields, for example:

i. Natural Language and Speech Processing

A prototype system EMILE [Adriaans 2002] is based on Adriaans' theoretical result on shallow grammars using categorical grammars [Adriaans 1992]. A system managing Deterministic Finite Automaton (DFA), transducers and probabilistic automata [Mohri 1997] is proposed for computational linguistics

applications.

As for speech, traditional models include *n*-grams [Jelinek 1998] and HMMs [Morgan 1995]. In the last decade, finite automata were proposed to model languages [Garca 1994]. More general models such as stochastic automata [Thollard 2000] and context-free grammars [Wang 2002] were also used in this direction.

In language translation applications, grammatical inference techniques are combined with domain knowledge, query [Vilar 1996] and data driven approaches [Oncina 1998] for transducer construction.

ii. Control

Grammatical inference is researched for modeling map learning in environments where robots can make moves based on their observations. Dean studied the learning of map where errors exist in the observation [Dean 1992].

Rieger constructed a stochastic model based on automaton which could be further used for navigation [Rieger 1995].

Luzeaux proposed to use grammatical inference as a learning model in intelligent control systems [Luzeaux 1996].

iii. Pattern Recognition

Finding patterns in bio-sequences is an interesting task. Pattern languages are proposed [Brazma 1998]. To classify DNA sequences, Wang et al. [J. T.-L. Wang 1999] proposed a method that first induces patterns, then classifies them by scoring. Modeling by context-free grammars allows the discovery of secondary structure [Abe 1997, Sakakibara 1994, Salvador 2002].

iv. Data Mining

Grammatical inference techniques are used for web mining by wrapping web sites with a more powerful class based on a subclass of formal language [Chidlovskii 2000]. For navigation behavior mining, Borges proposed a method to model the last N pages that a user visited with a stochastic grammar to analysis user preference [Borges 2000]. Giles combined grammatical inference and recurrent neural networks to deal with noises in time series mining [Giles 2001].

1.1 Motivation

Grammatical Inference (GI) is to identify grammatical models from generated samples. The grammar of programming languages, for example, could be such a grammatical model. Therefore, results in grammatical inference usually reflect theoretical foundation of computational learning process. Recently, due to the explosion of information and the need of efficient ways of searching and organizing, grammatical inference results are put into use

for application-oriented approaches. However, the general tasks of identifying any grammatical model from its generated strings have been proved to be impossible [Gold 1967]. Therefore, constrains must be given to some parts of the task: limiting the scope to a subset of language classes, equipping sample strings with additional information, designing variant models, or taking hybrid approaches. Since larger model size results in more space complexity, our work intends to research learnable subsets of regular languages and context-free languages under constrains and find methods to learn refined model with less size than existing methods.

1.2 Objectives

Traditionally, there are mainly three approaches in the grammatical inference field on regular languages by Angluin [Angluin 1981, Angluin 1982, Angluin 1987b]. Recent regular grammar learning methods use heuristic information to split and merge states in automata constructed from sample strings. All these approaches are proposed to construct automata for two language classes, *i.e.*, the model either accepts a string or rejects it. Though running multiple automata for multiple language classes is a straight forward extension, the resulting models are constructed with unnecessarily space cost. Our first objective is to extend the regular language learning methods to be capable of identifying a single automaton for multiple string classes to reduce the size of learned model.

We further use additional information in learning models. This leads us to adopt the model

of Cover Automaton [Kaneps 1990, Shallit 1996] for grammatical learning. Specifically we adopt Campeanu's approach [Campeanu 2001] for utilizing the information about length limit for the construction of minimal cover automata. Our objective is to leverage the length limitation to reduce the size of the learned model.

We then look at reducing the size of learned model from stochastic sequence data. In applications, models usually generate samples in a certain distribution. Therefore, models with stochastic information are often of more use, as they represent such distributions. The technique of Markov Chains is a main method for modeling of stochastic sequence data. In traditional framework of Markov Chains [Rabiner 1989], the models are able to capture stochastic information of sequences. In the first-order Markov Chains, only two-length sequence patterns are captured, while in the second-order Markov Chains, three-length sequence patterns are captured. In the same manner, k^{th}–order Markov Chains can and only can capture $(k+1)$-length patterns. However, capturing all such patterns requires exponential spaces. Our objective is to find methods to identify mixed k-th ordered Markov Chains from given string samples with an efficient model.

Alignment Based Learning (ABL) [Zaanen 2002a] is a practical and interesting algorithm for generating context-free grammatical rules from a given sample. Our next objective is to refine the learned model for improving the precision of context-free grammatical rules extracted from samples by reducing identified invalid rules.

In summary, our objectives are finding methods that learn from regular language samples with reduced model size and refining context-free language learning algorithm with reduced number of incorrectly identified rules and improved precision.

1.3 Contributions

We propose an extension method to reduce the size of resulting automata for regular language learning where samples in multiple language classes are given. We propose the use of our extension method for the following three approaches: learning regular languages from a representative sample and using membership queries [Angluin 1981], identifying regular sets using equivalence queries and membership queries [Angluin 1987b] and the inference of reversible languages [Angluin 1982]. We prove the correctness of the algorithms. Our algorithm based on the third approach is called *cl-ZR*. The extension of the third approach requires no query in the learning process and therefore can be tested through experiments. The results are based on the experiments on the house renting data extracted from emails within the public folder of the email server in Nanyang Technological University, which showed averagely 50% reduced running speed, 2.8 times reduced error rate and 20% reduced size than constructing separated automata for each language class.

We give an algorithm to construct the minimal cover automaton from sample strings of the target finite language. The algorithm is proved correct and efficient through experiments on the data set Splice-junction Gene Sequences Database

(http://www.ics.uci.edu/~mlearn/MLSummary.html). This method averagely reduced the size of the learned automata by 30%.

We present a construction method to reduce the stochastic model size greatly based on pruning method. In the experimental results on the data set Splice-junction Gene Sequences Database (http://www.ics.uci.edu/~mlearn/MLSummary.html), we show that by using combined longer sequencing information, the error rate is reduced by above 1.56 times with our method to Markov Chains Model.

In extending our work to context-free grammars, we are inspired by the alignment based learning. Alignment Based Learning (ABL) [Zaanen 2003] is a framework for learning context-free grammar rules from given positive samples. Alignment results are used as possible boundaries of context-free grammar rules. We propose the Profiled Based Alignment Learning framework. Methods, namely slot alignment, alignment profile, sentence similarity, are designed to refine the alignment during the learning process and therefore improve the results. We applied the methods on samples from the data set CHILDES [Bliss 1988, Carterette 1974, Chang 1998, Jones 1963, MacWhinney 2000]. Our method refined the average result with reduced 15.8 times error rate than ABL.

1.4 Organization

The organization of this thesis is as follows. Chapter 2 gives a review of literature in grammatical inference methods. The generalized model, classification automata, is discussed

in chapter 3. The learning of the alternate models, cover automata and recurrent neural networks for mixed *k*-th ordered Markov Chains, are discussed in Chapter 4 and Chapter 5 respectively. Profile based alignment learning is discussed in Chapter 6. Chapter 7 presents concluding remarks.

CHAPTER

2

Grammatical inference Methods

Over the past decades, researchers in these fields were brought together on conferences and workshops to share their ideas on grammatical inference. Good surveys on this subject include [Higuera 2005, Lee 1996, Sakakibara 1997].

2.1 Basic Learning Models

Before introducing the methods and results, we introduce some basic notations. When learning means inference from examples, the *examples* could be real numbers, points in a certain space, strings of symbols, Boolean assignments, and so on. A set X of possible examples is selected. The set X is defined as the *example space*. A *concept* c is a subset of X. A concept class C is a collection of concepts. An example $y \in X$ is called a *positive example* of concept c when $y \in c$, a *negative example* otherwise. A pair (y, (true if $y \in X$, false if $y \notin X$)) is called a *labeled example*. A *target concept* is the concept generating the learning samples. When we say that an unknown target concept is to be learned, it means that learning algorithms will output an approximation of the concept. The hypothesis space H of

a learning algorithm is the set of concepts, from where a computational representation of hypotheses from H is selected as the output of the algorithm. Since the choice of H and its computational representation will affect the learnability of C, we usually write *learnability of C in terms of H*. Given X, C, and H, along with their computational representations, a *learning algorithm A* has access to examples determined by some $c \in C$ to some distribution D. When A halts, its output will be a single concept from H.

There are three basic formal learning models: *identification in the limit* by Gold [Gold 1967], *PAC learning* model by Valiant [Valiant 1984] and *query learning* by Angluin [Angluin 1988]. The three models have different criteria and descriptions of learning.

2.1.1 Identification in the limit

In the *identification in the limit* model, an infinite sequence of examples of the unknown language L with grammar G is given to the learning algorithm M and the behavior of the algorithm after some finite time is used to describe the criterion of successful learning. Σ is the alphabet of the target language. The set of all strings over an alphabet Σ is denoted Σ^*. A complete presentation of grammar G is an infinite sequence of labeled examples $(y, l) \in \Sigma^* \times \{0,1\}$ such that $l = 1$ if and only if (iff) string y is in the language generated by grammar G, i.e. y is a positive example, and every string y of Σ^* appears at least once in some pair in the sequence. At each time t, the learning algorithm M receives an information unit (usually an example) i_t of a complete presentation of G, and outputs a hypothesis $H(i_1,\ldots,i_t)$. A

learning algorithm is considered to be successful if, after a finite amount of time, the hypothesis that the algorithm guesses is equivalence to *G* at some point, and stops changing when more examples are taken as input. If, for every complete presentation of the unknown grammar *G*, a learning algorithm *M* successfully learns *G*, then *M* is said to *identify G in the limit from complete presentations*. While infinite sequence of examples is not practical, identification in the limit model works as a basic model that learns from examples.

2.1.2 PAC Learning

In Valiant's seminal work [Valiant 1984], Probably Approximately Correct (PAC) learning model is defined. In PAC learning model, examples are generated to an unknown distribution *D*, and the goal of learning is to get results with high accuracy and high possibility. Formally, the definition is given as follows.

To approximate the target concept, a learning algorithm is provided labeled examples, generated randomly according to some unknown distribution *D* over *X*. Given a real number δ ($0<\delta<1$) and a real number ε ($0<\varepsilon<1$), then there is a positive integer $m_0 = m_0(\delta, \varepsilon)$ such that for any target concept $t \in H$, and for any probability distribution *D* on *X*, whenever the algorithm takes *m* examples ($m \geq m_0$) generated according to *D*, the learning algorithm will, with a probability at least $1-\delta$, output a hypothesis $h \in H$ that has probability at most ε of disagreeing with the target concept.

In other words, given enough number of examples, with a probability at least $1-\delta$, the error

rate of the hypothesis produced is less than ε when the learning algorithm is supplied with labeled examples. If such a learning algorithm exists, then *C* is *PAC learnable*.

If algorithm *A* PAC learns concepts from *C* in terms of *H* with respect to the class of all possible distributions on *X*, then we should say *A* PAC learns concepts from *C* in terms of *H*. The distribution-free requirement is that the learning algorithm works with respect to an arbitrary unknown distribution.

2.1.3 Query Learning

When human learns, the access to experimental results is sometimes useful. The same idea is applied to grammatical inference. In the *query learning* model, the learning algorithm has access to oracles given by a teacher that can answer certain types of questions about the target language. In some stages of the running process of the learning algorithm, query results can be used to aid the learning of the target grammar. There are two basic types of queries that are widely used in this field:

1) *Membership Query*

   The learning algorithm generates a string $w \in \Sigma^*$. The teacher (or, oracle) returns "yes" as the query result if *w* is a member of the language generated by the target grammar. Otherwise, the query result is "no".

2) *Equivalence Query*

The learning algorithm generates a grammar *G'*. *G'* is usually the grammar that the learning algorithm guesses. Then, the teacher (or, oracle) returns "yes" as the query result if the language generated by *G'* is equal to that by *G*, *i.e.*, *G'* is equivalent to *G*. Otherwise, the query result is returned as "no", and it will also return a counter-example.

Let $L(G)$ and $L(G')$ denote the languages generated by G and G', respectively. A *counter-example* is a string $w \in \Sigma^*$ such that, either $w \in L(G)$, $w \notin L(G')$, or $w \in L(G')$, $w \notin L(G)$. In other words, a counter-example is an evidence of difference between G and G'.

A formal definition of learning by equivalence queries is given below [Angluin 1988].

Definition 2.1 Learning by Equivalence Queries: Let R be a representation of a class of languages $\mathcal{L}$, and let $r(L)$ be any representation for a language $L \in \mathcal{L}$. An equivalence oracle for $L_* \in \mathcal{L}$ takes as input a $r(L) \in R$, it returns "yes" if L is equivalent to L_* (means that for any string $w \in \Sigma^*$, $w \in L$ iff $w \in L_*$, denoted $L \equiv L_*$), otherwise, it returns "no" with a string in the symmetric difference of L and L_*. A deterministic algorithm M exactly identifies R using equivalence queries in polynomial time iff there is a polynomial $p(.,.)$ such that for all L_*, when M runs with access to an equivalence oracle for L_*, at any point in its computation, M has used an amount of time bounded by $p(l, m)$, where m is the maximum length of any counter-example returned so far and l is the length of the shortest representation of L.

2.1.4 Learnability Results

Learnability Results are results that state whether certain language classes can be learned by learning models. Gold has presented main learnability results in [Gold 1978] which we briefly state below. When the learning algorithm is given positive and negative examples of the target language (the language to be learned), and each example appears at least once, then the learning algorithm is capable to identify the language. When only positive examples of the target language are given, then the identification is impossible for any super-finite class of languages. A super-finite class of languages is one that contains all finite languages and at least one infinite language, where a finite language is a language with finite number of different strings and an infinite language is a language with infinite number of different strings.

Definition 2.2 A *Deterministic Finite Automaton* (DFA) [Hopcroft 2001] is a quintuple $A=(\Sigma, Q, q_0, \delta, F)$, where Σ is the alphabet and Q are finite nonempty set of states, $q_0 \in Q$ is the starting state, $F \subseteq Q$ is the set of final state(s), and δ: $Q \times \Sigma \rightarrow Q$ is the transition function: $\delta(q_1, a) = q_2$ means that if the current state of the automaton is q_1 and the next input symbol is a, then the state will transit to q_2. δ can be extended from $Q \times \Sigma$ to $Q \times \Sigma^*$ by $\delta(q, \lambda) = q$ and $\delta(q, aw) = \delta(\delta(q, a), w)$, $q \in Q$, $a \in \Sigma$, $w \in \Sigma^*$. The language recognized by the automaton A is $L(A)=\{w \in \Sigma^* | \delta(q_0, w) \in F\}$. If L is $L(A)$ for some DFA A, then we say L is a *regular language (regular set)*. Let l be the length of the longest word(s) in the finite language L. A DFA A such that $L(A) \cap \Sigma^{\leq l} = L$ is called a *deterministic finite cover automaton*

(DFCA) of L. Let α be a string of terminals and/or nonterminals, the cardinality of α is denoted by $|\alpha|$.

Definition 2.3 A *Context-free Grammar* [Hopcroft 2001] (CFG) $G = (N, \Sigma, P, S)$, where N is the set of nonterminals, Σ is the alphabet, P is the set of productions, and $S \in N$ is the start symbol representing the language being defined. Let $A \rightarrow \alpha$ be a production of G, when G is understood in the context, then we say $\beta\alpha\gamma$ is derived from $\beta A\gamma$, denote $\beta A\gamma \Rightarrow \beta\alpha\gamma$. If β is derived from α by zero or more steps, the derivation is denoted as $\alpha \Rightarrow^* \beta$. The *language of G*, denote $L(G)$, is the set of terminal strings that can be derived from the start symbol. Formally, $L(G) = \{x \mid S \Rightarrow^* x \text{ and } x \in \Sigma^*\}$. If a language L is the language of some context-free grammar, then we say L is a *Context-free Language* (CFL).

Deterministic Finite Automaton is found to be learnable in the limit [Gold 1978]. However, identifying DFA in polynomial time is a computationally difficult problem [Gold 1978], the problem of finding the smallest DFA consistent with a given set of strings is proved to be NP-hard. Even when the target automaton only has two states, or even when only a very small fraction of all the strings up to length n (the size of the target language) is absent, the problem is NP-hard [Angluin 1978]. Angluin designed a technique of *approximate fingerprints* for studying learnability for query learning models [Angluin 1990]. Using this technique, Angluin has shown that there is no algorithm using only equivalence queries identifies in polynomial time for the classes of deterministic finite automata, nondeterministic finite automata, context-free grammars, or disjunctive or conjunctive normal form Boolean formulas. Angluin proved in [Angluin 1987b] that regular languages

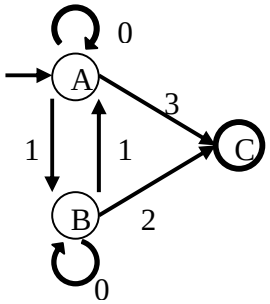
could be identified with only a polynomial amount of queries. Context-free grammars are found to be not learnable based on cryptographic assumptions [Angluin 1991].

2.2 Inference of Regular Languages

There are mainly three approaches for grammatical inference on the learning of regular sets. One is learning regular sets from representative samples and using membership queries [Angluin 1981]. The second one is identifying regular sets using equivalence queries and membership queries [Angluin 1987b]. The third one is the inference of reversible languages [Angluin 1982].

2.2.1 Learning from Representative Samples and Membership Queries

A *live-complete set* for regular language L is any finite set of strings P such that for every live (reachable and producible) state q of A=DFA(L) there exists a string $u \in P$ such that $\delta(q_0,u) = q$. A *representative sample* of L is any finite subset S of L such that for every live (reachable and producible) transition $\delta(q,b)$ of A, there exists a string u in S that uses $\delta(q,b)$. An example of representative sample is shown in Figure 2.1.



{0113, 102} is a representative sample, but {03, 102} is not a representative sample, since the transition from B to state A is not used in generating {03, 102}.

Figure 2.1 Representative sample

A new symbol d_0 is used to denote dead state. Let $P'=P\cup\{d_0\}$. Let T' be the set of all elements of P' and all elements of $f(u,b)$ (elements accessed by (u,b)) for all $(u,b)\in P\times\Sigma$. Let $T=T'-\{d_0\}$.

Algorithm 2.1 Algorithm ID

//Initialization

- $E_0(d_0)$=null set, v_0 (the *0*-th distinguishing string) =λ;
- For each $u\in T$,

  -- if membership query about string u returns "yes",

  let $E_0(u) = E_0(u) \cup\{\lambda\}$.

//Loop

- Once $E_i(u)$ has been constructed for all $u\in T'$, search for a pair of elements $u,v \in P'$, and a symbol $b\in U$, such that $E_i(u)=E_i(v)$ but $E_i(f(u,b) \neq E_i(f(v,b)$.
- If find such a pair,

  -- Choose some string $w\in E_i(f(u,b)\oplus E_i(f(v,b)$, and define v_{i+1} to be bw.

  -- Set $E_{i+1}(d_0)$ to null set.

  -- For each $u\in T$,

  -If query about uv_{i+1} returns "yes", $E_{i+1}(u) = E_i(u) \cup\{v_{i+1}\}$.

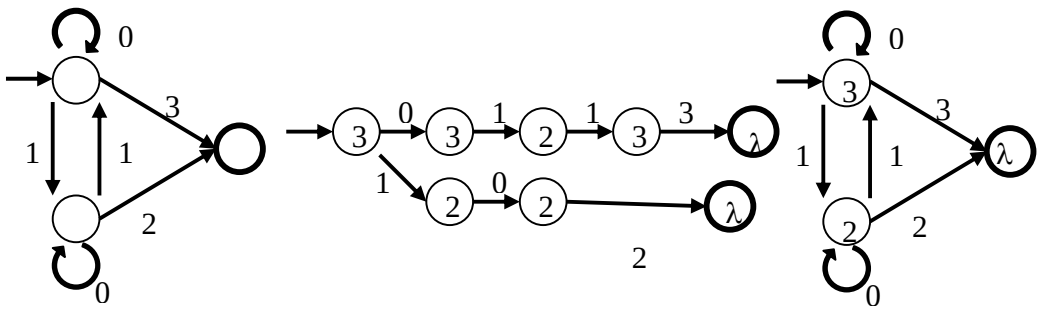
  Else, merge all states with the same $E(u)$ functions, output.

//End of algorithm

The algorithm ID [Angluin 1981] first uses all the prefixes of representative sample strings as accessing strings, and creates each a state. Suppose for the target automaton in Figure 2.2 (a),

the representative sample *S* is {0113, 102}. Algorithm ID creates nodes for the prefixes {λ, "0", "01", "011", "0113", "1", "10", "102"}. Since the sample set is representative sample set, all the nodes and links are already in the DFA constructed. Then, the algorithm merges nodes and transitions to get the target DFA. A *distinguishing string* (or ending string) a state is a string that can be used from the state to transit to the final state of the target DFA. For example, one distinguishing string of the state corresponding to "01" in Figure 2.2 (b) is "2" as in the target automaton, "012" is accepted, i.e., transiting to the final state. The *distinguishing string set* of a state is the set of all distinguishing strings of a state. Algorithm ID is based on the idea that states with the exact same distinguishing string set must be the same state in the target automaton, as they are undistinguishable. Then the algorithm uses membership queries to find out the distinguishing string set of each state. Starting from the empty string λ being the distinguishing string, membership queries are proposed on λ, "0", "01", "011", "0113", "1", "10", "102", with "0113" and "102" being true. Hence, λ is included in the distinguishing string set of the states corresponding to "0113" and "102". Then, for each node and each symbol in the alphabet, a membership query is asked. Based on membership query results, transition information is revealed. For example, the node can be reach by "01" in Figure 2.2 (b) generates 4 membership queries: "010", "011", "012", "013", among which only the query "012" returns a true result. This means that the node corresponding to "01" can only transit to the final state by symbol "2". Hence "2" is included in the distinguishing string set of the state of "01". The marking result is as shown in Figure 2.2 (b). Next, algorithm ID uses the connection of all symbols in the alphabet and all known distinguishing strings as new candidate distinguishing strings. For example, the

distinguishing strings used after the step shown in Figure 2.2 (b) are “02”, “12”, “22”, “32”, “03”, “13”, “23” and “33”. Hence, there is a partition of states where states with the same distinguishing string set are in the same partition subset. Such steps are repeated until the partition of states no longer changes. Then, the states in the same partition subset are merged as one single state, and the resulting automaton is constructed. The resulting automaton of the example used is shown in Figure 2.2 (c).



(a). target automaton (b) marking of states (c) resulting automaton

Figure 2.2 A running example of algorithm ID

With the algorithm ID, a learnability result is presented as follows.

Theorem 2.1 The class of deterministic finite automata can be identified in polynomial time from a representative sample and using membership queries [Angluin 1981].

2.2.2 Learning by Membership and Equivalence Queries

The details of the algorithm identifying regular sets using equivalence queries and membership queries can be found in Dana Angluin's work [Angluin 1987b]. A variant method that based on the same idea but in a more understandable presentation can be found in [Kearns 1994]. The idea is to use the counter-example strings returned by equivalence queries to maintain a tree of distinguishing strings, and thus find more states in the target automaton.

Suppose S is the access string set, an access string is denoted by s. D is the distinguishing string set, a distinguishing string is denoted by d. T is the classification tree of distinguishing strings and access strings. M is the target automaton.

M' is the learned automaton, and is updated during the algorithm. γ is the counter-example string, which will generate different accepting results in the target automaton and the guessed one. $\gamma[i]$ is an array of the first i elements of γ. γ_j is the j'th 'letter' of γ.

The procedures for constructing DFA by membership queries and equivalence queries are given in Algorithm 2.2.

Algorithm 2.2 Algorithm Learn-Automaton [Kearns 1994]:

Procedure Sift(*s*,*T*):

- Initialization: set the current node to be the root node of *T*
- Main Loop:

  --Let *d* be the distinguishing string at the current node in the tree.

  --Make a membership query on *sd*. If *sd* is accepted by *M*, update the current node to be the right child of the current node. Otherwise, update the current node to be the left child of the current node.

  --If the current node is a leaf node, then return the access string stored at this leaf. Otherwise, repeat the Main Loop.

Procedure Tentative-Hypothesis (*T*):

- For each access string (leaf) of *T*, create a state in *M'* that is labeled by that access string. Let the start state of *M'* be the state λ.
- For each access state *s* of *M'* and each $b \in \{0,1\}$, compute the *b*-transition out of state *s* in *M'* as follows:

  --*s'* $\leftarrow$ **Sift**(*sb*, *T*)

  --Direct the *b*-transition out of state *s* to state *s'*
- Return *M'*

Procedure Update-Tree (γ ,*T*):

- For each prefix $\gamma[i]$ of γ:

  --$s_i \leftarrow$**Sift**($\gamma[i]$, *T*).

  --Let $s_i' = M'[\gamma[i]]$.

- Let *j* be the least *i* such that $s_i \neq s_i'$.
- Replace the node labeled with the access string s_{j-1} in *T* with an internal node with two leaf nodes. One leaf node is labeled with the access string s_{j-1} and the other with the new access string $\gamma[j\text{-}1]$. The newly created internal node is labeled with the distinguishing string $\gamma_j d$, where *d* is the correct distinguishing string for s_j and s_j'.

Main Procedure

- Initialization:

  --Do a membership query on the string λ to determine whether the start state of *M* is accepting or rejecting.

  --Construct a hypothesis automaton that consists simply of this single (accepting or rejecting) state with self-loops for both the 0 and 1 transition.

  --Perform an equivalence query on this automaton, let the counterexample string be γ.

  --Initialize the classification tree *T* to have a root labeled with the distinguishing string λ and two leaves labeled with access string λ and γ.

- Main Loop:

  --Let *T* be the current classification tree.

  --*M'* $\leftarrow$ **Tentative-Hypothesis**(*T*).

  --Make an equivalence query on *M'*. If it is equivalent to the target then output *M'* and halt. Otherwise, let γ be the counterexample string.

  --**Update-Tree**(γ, *T*).

- Repeat Main Loop.

For example, the target automaton is the one shown in Figure 2.2 (a). The learning process is in the following steps as shown in Figure 2.3. First of all, an automaton is created with the start state as the only state. Transitions of all symbols in the alphabet, in this case is '0', '1', '2', '3', are created self-loops to this single state. Then membership query is done on the empty string. Since the result is 'rejected', the start state is not a final state. After that, an equivalence query is submitted to see if the machine is the target one, and suppose a counter example "1103" is returned by the oracle as it can be accepted by the target automaton but gets rejected by the current one. Then the classification tree *T* is initialized as a simple tree with the root node marked with the empty string as the distinguishing string, empty string as its left child and the counter example "1103" as its right child. In the algorithm, accepted strings are put at the right-hand side while rejected ones are put at the left-hand side. In the example, "1103" connecting with empty string is accepted and, therefore, is put at the right-hand side. Based on the tree, one can know that there is one group of states that cannot reach the final state set with an empty string starting from them, and the start state is one of them while the other group of states that can reach the final state set with an empty string, and one of such state can be accessed by "1103". The learned automaton is updated based on the classification tree. The two known state sets are constructed as two states in the learned automaton. The transitions between the two sets of states are discovered by the Sift procedure based on membership query: the accessing strings of the states, which indicate the empty string and "1103" in the example, are extended with all symbols in the alphabet. Here, the strings resulted by extension are "0", "1", "2", "3", "11030", "11031", "11032" and "11033". Sift procedure is used to find out which state should set the extended strings

leading to, and therefore, the transitions of the known state sets are learned. For example, "11031" is given to the Sift Procedure. Starting from the root node, "11031" is connected to the empty string carried by the node. The connection result "11031" is submitted to a membership query, and the result is "rejected". By this piece of information, it is learned that the transition of "1" from the state "1103" should point to the state of the empty string. Similarly, other transitions are determined. Then suppose the next counter example is "012". This counter example is used to update the tree. In the Update Tree procedure, Sift is used in each prefix of the counter example to find the shortest prefix that leads to the difference between the learned automaton and the target one. In the example the prefix is "012". This means that "01" leads to the same state set in the target automaton and the learned on, while "012" leads to different states in the two automata. Therefore, the state set that "012" leads to should be split, as there is confliction in it. In the tree, the corresponding node is split into two nodes, one is the old one marked with the empty string, and the new one is "01". The root node of the split two nodes is "2", meaning that, in the state set in the learned machine, there is a set of state that can reach the final states by a "2" transition and there is a set cannot. The learned automaton is updated accordingly and the above steps are repeated until the equivalent query suggests that the learned automaton is the same as the target one.

Based on Algorithm 2.2, we have the following learnability result:

Theorem 2.2 The class of deterministic finite automata can be identified in polynomial time using equivalence queries and membership queries [Angluin 1987b].

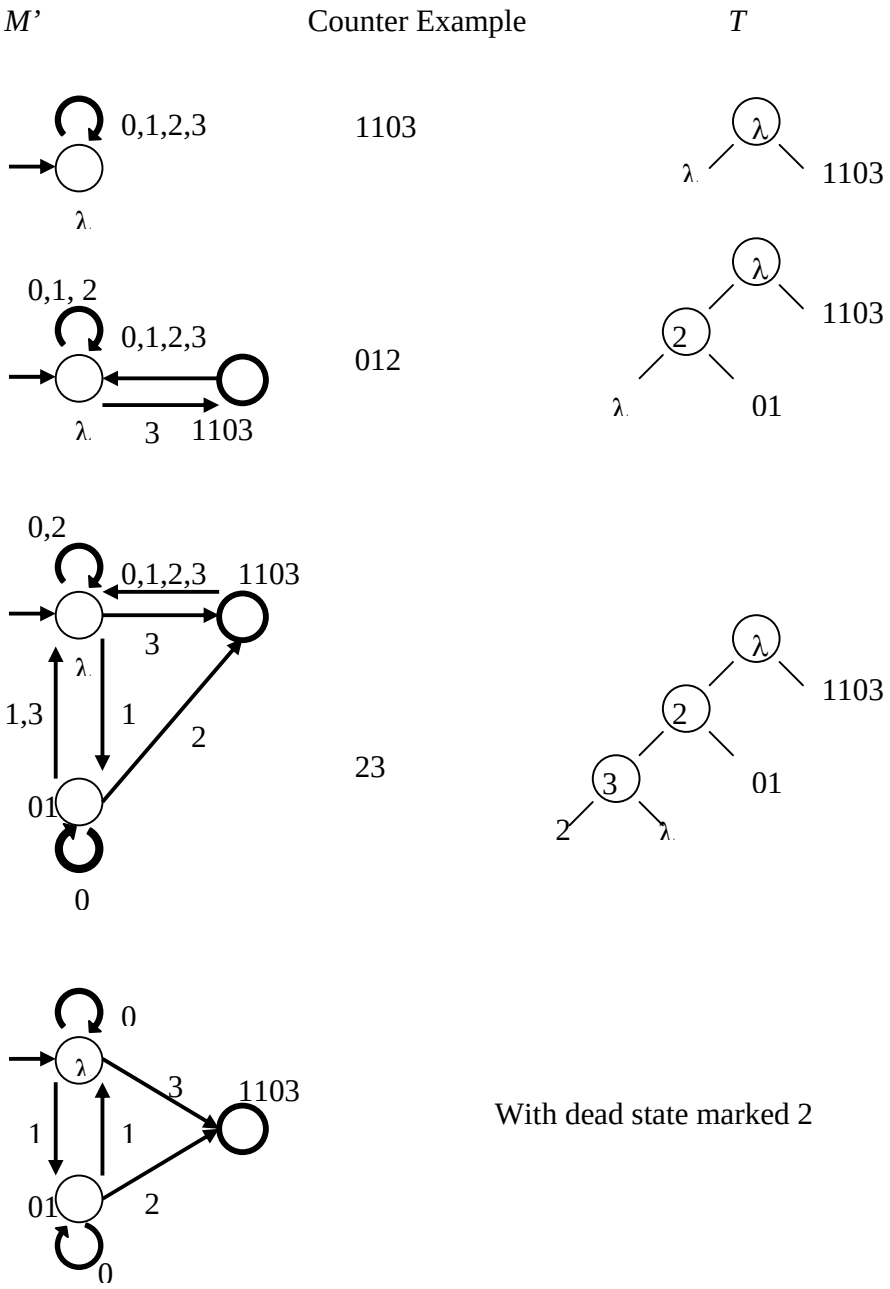


Figure 2.3 A running example learning by equivalence queries and membership queries

2.2.3 Learning Reversible Languages

A *0-reversible language* is a language, which can be accepted by a DFA with only one start state and one final state, and when all arrow in the transition graph are reversed, it is also a DFA. A *1-reversible* language is a language that can be accepted by a deterministic automaton, and its reversed automaton is deterministic with lookahead 1, *i.e.*, for any terminal a and states q_1, q_2, q_3 and any terminal string α, $|\alpha|=1$, if we have $\delta(q_1, \alpha a) = q_2$ and $\delta(q_1, \alpha a) = q_3$, then q_2 and q_3 must be the same state.

Similarly, a k-reversible language is a language that can be accepted by a deterministic automaton, and its reversed automaton is deterministic with lookahead k, *i.e.*, if any state C can reach states A and B with the same transition in length k, then A and B must be the same state. For example, when k = 3, if there is a state C can reach state A by transition of "010" and state C can also reach state B by transition of "010", then by looking ahead 3 symbols in the reversed automaton, state C can reach both state A and B with the same transition of "010". Then, either state A and B must be the same state or the language accepted by the automaton is not a 3-reversible language. The language class of all k-reversible languages (k = 0,1,2…) is called *reversible language*.

Angluin introduced the above k-reversible languages as a series of subclasses of regular languages [Angluin 1982]. In her paper, it is shown that a characteristic sample is a

sufficient condition for identification from positive samples for *k*-reversible languages. A *characteristic sample* of an automaton *A* of a *k*-reversible language is a finite sample $S \subset L(A)$ such that $L(A)$ is the "smallest" *k*-reversible language that contains *S*. It is shown that any characteristic sample is a representative sample.

Definition 2.4 Let *L* be a finite language, define the *Prefix Tree Acceptor* for *L*, PT(*L*) =(Σ, *Q*, q_0, δ, *F*), where Σ is the alphabet, *Q*= Pre(*L*), the set of all prefixes of *L*, $q_0=\{\lambda\}$if $L\neq\phi$, $q_0=\phi$ if $L=\phi$, *F*=*L*, $\delta(u, a)=ua$, $u, ua \in Q$.

For example, let $L=\{a, aa, aaa, baa\}$, PT(*L*) is shown in Figure 2.4.

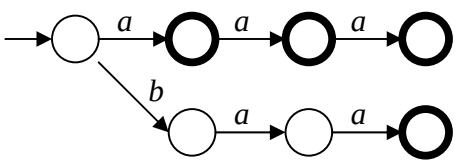
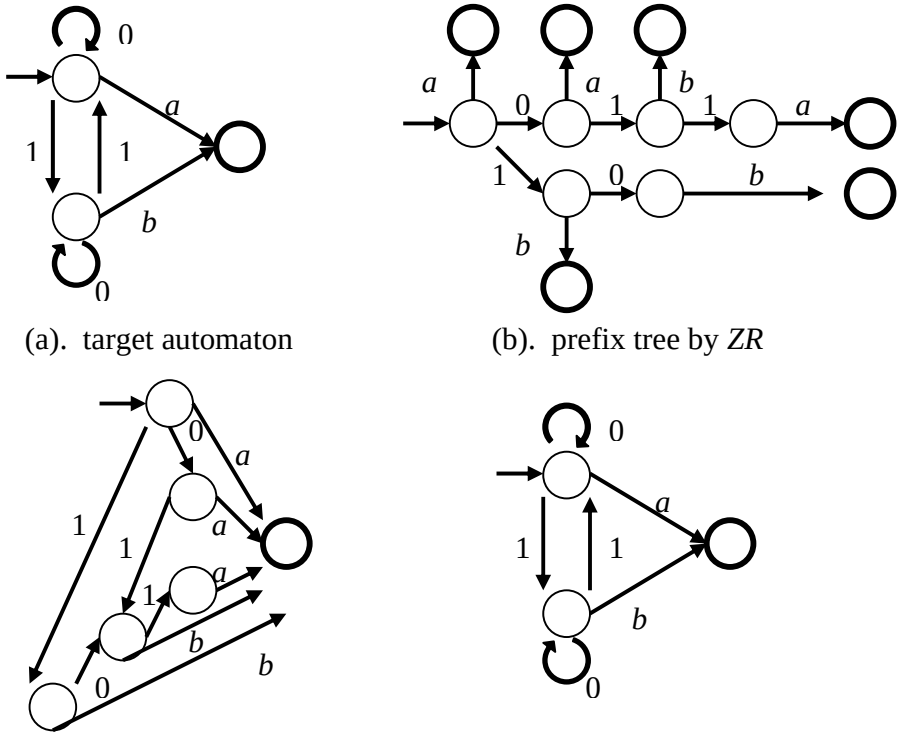


Figure 2.4 Prefix tree acceptor for *L*

The algorithm 2.3 (algorithm *ZR*) [Angluin 1982] first constructs the prefix tree acceptor of the given positive strings for a given language, and then it uses merge to satisfy the condition of zero-reversible language acceptor, and then get an automaton that accepts the target language. Suppose we have the training sample $S = \{a, 0a, 01b, 011a, 1b, 10b,\}$. We use this sample set on *ZR*. Firstly, *ZR* constructs the prefix acceptor for the given sample *S*, as showed in Figure 2.5 (b). Then it merges the final states (Figure 2.5 (c)), where 3 states have

transitions '*a*' to the final state and 2 states have transitions '*b*' to the final state. Therefore, the automaton in Figure 2.5(c) is not 0-reversible. The 3 states with transitions '*a*' to the final state are then merged into one state, and the 2 states with transitions '*b*' are merged into another state. As a result of merging, the automaton shown in Figure 2.5(d) is 0-resversible, *i.e.* there are no different states with the same transition pointing to the same state. Therefore, this resulting automaton is the target automaton.



(a). target automaton (b). prefix tree by *ZR*

(c). merge final states (d). resulting automaton by *ZR*

Figure 2.5 A running example of ZR algorithm

Algorithm 2.3 Algorithm *ZR*:

//Initialization

- Construct the prefix tree PT(*S*) of the given sample *S*.
- Acceptor *A*=PT(*S*)

//loop

- From all states

  Find B_1 and B_2 such that $\delta(B_1, a) = \delta(B_2, a) = B_3$
- If no such B_1 and B_2 exist, output and exit.

  Else merge B_1 and B_2

//end

The algorithm *ZR* [Angluin 1982] is to learn a minimal *0*-reversible language acceptor from sample set *S*, while the algorithm learning a minimal *k*-reversible language acceptor is a generalized version of algorithm *ZR*.

Theorem 2.3 [Angluin 1982] The class of *k*-reversible language, for *k*=0,1,2,…, can be identified in the limit from positive presentation.

2.2.4 Learning by Merging

A generalizing algorithm *regular positive and negative inference* (RPNI) is proposed by Oncina and Garcia [Oncina 1992]. RPNI algorithm takes a characteristic set as input and generates the smallest DFA consistent with the set. Here, a characteristic set is a sample set with this property: in the Prefix Tree Acceptor constructed from the set. If any states can be

merged in the Prefix Tree Acceptor, they must be the same state in the target DFA. RPNI is proved to converge with large number of strings as input. Dupont et al. use lattice to represent the search space of DFA [Dupont 1994]. The method enables the proof of convergence for other DFA inferring algorithms. However, traversal of the lattice is expensive in complexity. Lang [Lang 1992] has shown that this method is not practical.

In the Abbadingo competition [Lang 1998], a learner was given a set of sample strings consisting of both positive and negative examples labeled by the unseen target DFA, and was required to label a set of unlabeled testing strings. The string sets were drawn from uniform random distributions. One of the winner methods is *evidence driven state merging* (EDSM) method [Lang 1998]. The method starts with Prefix Tree Acceptor, marking its root (the starting state) as a red node. Then the method goes through the following steps until all nodes are marked red: marking every non-red node can be reached with one transition from the red node set as a blue node. Based on heuristic scoring, the pair of red node and blue node with the highest score is merged or the blue nodes are marked red if none of the pairs get sufficient score. The heuristic scoring is based on the number of the unchanged labels of the training sample by the resulting DFA that merges the nodes pair. A data driven heuristics method, where the score is based on information learned, is given by de la Higuera et al. in [Higuera 1996]. EDSM based methods are the most effective methods for complex regular grammars inference. The merges in such methods starts from the starting state, therefore, when the training strings are sparse, the methods tend to select the wrong merges early on and result in a model far from the target one[Cicchello 2002].

Parekh [Parekh 1996, Parekh 1998] proposed an incremental algorithm based on Angluin 's algorithm ID [Angluin 1987b]. The algorithm starts from any sample string set instead of a representative string set used in algorithm ID. Therefore, each time the resulting DFA is a part of the target DFA. On each new string that conflicts with the learned DFA, membership queries are used to create new nodes in the learned DFA. Therefore, the algorithm learns the target DFA incrementally.

2.3 Stochastic Inference

Stochastic modeling is rapidly growing for applications like natural language processing, speech recognition, and bioinformatics. A stochastic grammar has a probability for each production, and assigns a probability to each string that it generates. Hence, the grammar defines a distribution on the set of strings in the language.

Definition 2.5 *G* is *stochastic* iff there is an assignment of weights, each weight being between zero and one inclusive, to the right-hand sides of productions such that for every nonterminal *A*, the weights of the right-hand sides of all the productions with *A* on their left-hand sides sum to 1.

The learning of stochastic grammars from examples has two sub problems, one is to learn the structure of the grammar and the other is to estimate probabilistic parameters of the grammar. Estimating probabilistic parameters has efficient algorithms, which are based on expectation-

maximization. There are forward-backward algorithm for HMMs [Rabiner 1989] and inside-outside algorithm for SCFGs [Baker 1979, Lari 1990]. Both methods increase the likelihood of the training sample and finally reach a local maximum. Therefore, the goodness of the result and the convergence property depend on initialization.

In the learning of structure, Solomonoff has presented in his work [Solomonoff 1959, Solomonoff 1964] an encoding method that convert grammars into strings, which will enable further assigning of *apriori* probabilities to strings and grammars in the same way. He also suggested a way to search the grammar space: based on current grammar at each step, finding nearby grammars recursively with higher goodness value.

Horning [Horning 1969, Horning 1972] gave an algorithm that learns SCFGs in the limit with probability one from stochastic data (data derived by an SCFG). He also presented an idea of assigning *apriori* probabilities to grammars using a stochastic grammar for grammars. Other enumerative methods, which search *all* grammars with complexity less than a certain threshold, were presented by Wharton [Wharton 1977] and Van der Mude and Walker [Mude 1978]. A hill-climbing algorithm [Cook 1976] assigned a complexity measure for strings that has the following property: for strings of the same length and containing the same number of distinct symbols, if in a string all the symbols have the same frequency, the string is said to have the greatest complexity. Therefore, like the definition of entropy, the more random a string appears, the higher its complexity. Unfortunately, all the above algorithms are computationally inefficient.

In traditional framework of Markov Chains [Rabiner 1989], the models are able to capture stochastic information of sequences. In the first-order Markov Chains, only two-length sequence patterns are captured, while in the second-order Markov Chains, three-length sequence patterns are captured. In the same manner, k^{th}–order Markov Chains can and only can capture $(k+1)$-length patterns.

Neural networks are widely used on learning tasks [Eren 1997a, Eren 1997b]. Many approaches have been proposed to learn sequential information by neural networks, because neural networks have been proved to have strong representative power and believed to be able to capture sequential information. To achieve this, the neural networks need feedback loops that remember context information and are used to determine output while scanning on each symbol of the input sequences. This results in a general class called recurrent neural networks (RNNs) [Elman 1990, Giles 1992, Omlin 1996].

Rodriguez [Rodriguez 2001] found that simple recurrent networks do not generalize well on very large test sets. Boden and Wiles [Boden 2000] studied the learning ability of recurrent neural networks for simple context sensitive language $\{a^n b^n c^n\}$ and found that simple recurrent neural networks fail to generalize when n is greater than 13. Similar problems were also studied by J. Schmidhuber and F. A. Gers and D. Eck [Gers 2001a, Gers 2001b, Schmidhuber 2002], using "Long Short-Term Memory" (LSTM) recurrent networks, which is introduced by Hochreiter and Schmidhuber [Hochreiter 1997]. Their result shows that

though the LSTM does not generalize for all sizes, it exhibits excellent generalization performance, which is much better than simple recurrent networks.

Michael Husken and Peter Stagge presented a new approach to classify time series using recurrent neural networks in [Husken 2003]. In their recurrent neural networks, the internal dynamics is organized such that the output of the classification nodes represents the class of the time series fed into the network.

2.4 Inference of Context-free Languages

Since context-free languages are much more expressive, grammatical inference of context-free languages are important. Main approaches have been made, including learning by more helpful information, such as negative examples or structural information, and learning sub-classes of context-free languages.

2.4.1 Learning Using Helpful Information

The idea behind this approach is to present helpful information together with the string to the learning algorithm. One main approach is to use structural information. The point is that sometimes we are also interested in the derivation tree of the strings according to some CFL grammar.

Definition 2.6 *G* is an *operator* grammar iff no production has adjacent nonterminals in its right-hand side, *i.e.*, there are no productions of the form $A \rightarrow \alpha BC\beta$. *G* is an *operator precedence* grammar iff *G* is an *operator* grammar, and there are three binary relations *E*, *Y*, and *T* on Σ such that

(1) The relation *E* (equal in precedence) holds between all terminals that are adjacent or separated by a nonterminal in a derivation string.

(2) The relation *Y* (yield precedence) holds between the terminals preceding a handle and the leftmost terminal of the handle.

(3) The relation *T* (take precedence) holds between the rightmost terminal of a handle and the terminal immediately following the handle.

(4) *E*, *Y*, and *T* are pairwise disjointing.

A way to present structural information in a string form is called *parenthesis grammar* [McNaughton 1967]. For a context-free grammar *G*, the corresponding parenthesis grammar (*G*) is constructed by changing each production from $A \rightarrow \alpha$ to $A \rightarrow (\alpha)$ (suppose that parenthesis are not among the symbols in the Σ of *G*). An incremental algorithm for identifying operator precedence grammars in the limit from positive parenthesized data is presented by Crespi-Reghizzi [Crespi-Reghizzi 1971].

Structural data can be represented by *skeletons*, in which the nonterminal labels are removed from the derivation trees [Levy 1978]. Under this form of representation, they are exactly the set of trees accepted by Skeletal tree Automata (SA), which is a special kind of finite tree

automata with removed non-leaf labels. A finite tree automaton *A*, on a tree *T* as input, first assigns states to the leaves of *T*; later, it moves up the tree by assigning each state to each node on the basis of the states of the node's children according to the productions. If it reaches a final state at the root of *T*, it is said that *A* accepts *T*. Therefore, the problem of identifying CFLs from structured strings is reduced to the problem of identifying an SA.

By extending Angluin's inference algorithm ZR [Angluin 1982] to SAs, Sakakibara presented a method for learning context-free languages from structural data in polynomial time [Sakakibara 1992]. A reversible context-free grammar is a CFG *G* such that (1) $A \rightarrow \alpha$ and $B \rightarrow \alpha$ in the production set implies that $A=B$, and (2) $A \rightarrow \alpha B \beta$ and $A \rightarrow \alpha C \beta$ in the production set implies that $B=C$, where *A*, *B*, and *C* are nonterminals, and α, β are the star set of the union of nonterminals and terminals.

Theorem 2.4 [Sakakibara 1992] The class of reversible context-free grammars can be identified in the limit from positive presentation of structured strings provided that the structured strings are generated with respect to a reversible context-free grammar for the unknown context-free language.

From the above theorem, we can see that the whole class of CFGs cannot be identified from positive presentation of structured strings.

Informally, the idea of the algorithm of learning CFGs from structured strings is: for each structured string, according to the parenthesizes, the algorithm extracts productions that form the derivation procedure, and assign states for them. If the algorithm finds somewhere in the production set the following condition are satisfied:

(1) $A \neq B$, $A \rightarrow \alpha$ and $B \rightarrow \alpha$, or

(2) $C \neq B$, $A \rightarrow \alpha B \beta$ and $A \rightarrow \alpha C \beta$.

If condition (1) is satisfied, the algorithm will merge states *A* and *B*, if condition (2) is satisfied the algorithm will merge states *B* and *C*.

There are also other algorithms in this approach, such as an algorithm being able to identify SAs in polynomial time by structural membership and structural equivalence queries[Sakakibara 1990] and another algorithm similar to the one described above but needing "suitably selected" samples [Fass 1983].

2.4.2 Learning Subclasses of CFLs

Commonly, researchers avoid the non-learnability described in Gold's theorem by learning subclasses that do not contain all finite languages.

Definition 2.7 *G* is *k*-bounded iff the right-hand side of every production contains at most *k* nonterminals.

k-bounded CFGs are shown to be identifiable in polynomial time using equivalence queries and nonterminal membership queries [Angluin 1987a]. Nonterminal membership queries give out a string *w* and a nonterminal *A* to the oracle; the oracle gives answer "yes" if *w* is derivable from *A* and "no" otherwise. Hence, from the nonterminal membership queries, the learning algorithm has access to the structure of the target grammar.

Definition 2.8 *G* is a *simple deterministic* grammar iff all productions are of the form $A \rightarrow a\alpha$ where $\alpha \in N^*$, (*N* is the set of nonterminals, *N** is all possible strings over *N*), and $|\alpha| \leq 2$, and if $A \rightarrow a\alpha$ and $A \rightarrow a\beta$ in *P* then $\alpha = \beta$. *Simple deterministic languages* are languages accepted by a 1-state deterministic pushdown automaton with empty stack.

The above method is extended by Ishizaka to *simple deterministic languages* (SDLs) [Ishizaka 1990]. In Ishizaka's method, nonterminal membership queries are no longer needed; instead, the algorithm uses *extended equivalence queries*. In such a query, the algorithm gives out a grammar *G* that does *not* have to be a grammar for a SDL to the oracle. The oracle gives answer "yes" if the target grammar is equivalent to *G* and "no" otherwise. Extended equivalence queries are weaker than nonterminal membership queries, because they don't convey structural information. However, since equivalence of CFGs is undecidable, the oracle answering the query must be quite powerful.

Yokomori gave a polynomial algorithm for learning SDLs that only proposes grammars of SDLs, but his algorithm required *prefix membership queries* and *derivative membership*

queries [Yokomori 1988]. Prefix membership queries propose a string w, and the answer is "yes" if w is a prefix of any string in the target language L_*. Derivative queries propose two pairs of strings (u, v) and (u', v'), and the answer is "yes" iff $\{w \mid uwv \in L_*\} = \{w \mid u'wv' \in L_*\}$.

Definition 2.9 G is *linear* iff all productions are of the form $A \rightarrow uBv$ or $A \rightarrow u$ (with u, v as strings of terminal symbols). G is *even linear* if all productions are of the form $A \rightarrow uBv$, where $|u| = |v|$.

Takada has shown a way by which learning *even linear languages* (ELLs) can be reduced to the problem of learning regular languages [Takada 1988]. He used a control set to describe the possible derivations of strings in $L(G)$. It was proved that for any alphabet, there is a *universal* even linear grammar G_0 that generates any ELL over that alphabet with a control set that is regular. Hence, to learn an ELL, it should first learn the corresponding regular control set. The translation between control strings and strings in the language and between control sets and even linear grammars can be done in polynomial time, which means that equivalence and membership queries proposed by a regular language learning algorithm can be rewritten in terms of the corresponding even linear language. Extended result to linear languages can be found in [Takada 1987], a similar result was given in [Makinen 1990].

2.5 Application Oriented Methods

Besides studying learnability of different language models, researchers have proposed methods for real applications. This section introduces some of them.

2.5.1 SEQUITUR

A compressing method, called SEQUITUR, was described in [Nevil-Manning 1997]. The idea is to build a new production rule for the highest observed pair in the given text sample. After applying the new production rule, the process is repeated until no more pair can be found. The SEQUITUR algorithm searches the rules and replacement of pairs efficiently by merging rather than starts over for each pair.

2.5.2 Lattice and Squeeze

In [Pang 2003], Bo Pang, Kevin Knight and Daniel Marcu described a syntax-based algorithm that automatically builds finite state automata, called word lattice, from semantically equivalent translation sets. They first parsed the given sentences into a parse forest, and then merged them. The merging rule is that if two parse trees have the same root, and 1-level subtree from the same root also matches, then these children (of the matching roots) are merged (keeping subtree structure intact). For example, in Figure 2.6, “NP” in tree 1 and tree 2 are merged. Then, they traverse the parse forest top-down and create alternative paths for every merged node. As shown in Figure 2.6, in the word lattice, a word is

associated with each edge. By this way, different paths from "BEG" to "END" node form different paraphrases of the same sentence.

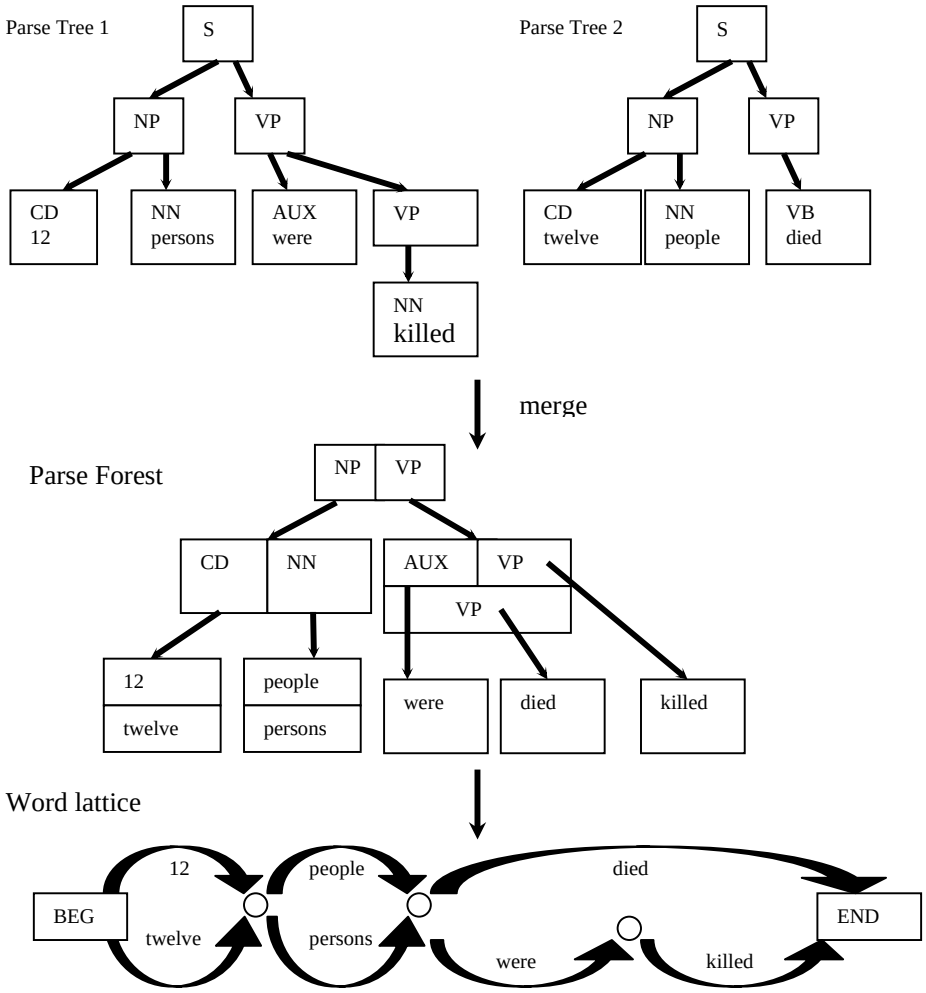


Figure 2.6 Top-down merging and FSA extraction of the syntax-based alignment algorithm

2.5.3 Alignment Based Learning

Alignment Based Learning (ABL) [Zaanen 2002a, Zaanen 2002b, Zaanen 2003] is based on alignment information. In ABL, the framework first does pairwise alignment for each pair of the input sentences, finding equal parts and unequal parts between them. A subsentence is also called as a constituent.

Pairwise alignment is an arrangement of two sequences which shows where the two sequences are similar, and where they are different. A good alignment shows the most significant similarities, and the least differences. Usually a score is assigned to an alignment, called *alignment score*, to measure the goodness of an alignment. Scoring scheme is usually defined on the pairing of different constituents and gap penalty for shifts in the alignments. As an example, consider following two sentences:

Monsters like tuna fish sandwiches.

All monsters like to fish.

For the above sentences, a possible alignment is as follows:

	Monsters like	*tuna*	*fish*	*sandwiches.*
All	*monsters like*	*to*	*fish*	.

Words that are located above each other and that are equal in the alignment, like the two "*Monster*"s, are called matches. The shifts caused by insertion or deletion are called *gaps*. In

alignment based system, more gaps means less similarity. Words that are located above each other that are unequal in the alignment are called *substitutions*. In an alignment, if there is a substitution, then the two subsentences are said to be *aligned in the same slot*, where the *slot* denotes where the subsentences are located in the alignment. For example, the subsentences "*tuna*" and "*to*" are aligned in the same slot which is shown by the brackets.

Monsters like [*tuna*] *fish sandwiches*.

All monsters like [*to*] *fish*.

In the first step of ABL, pairwise alignments are performed on sentences. In this step, the matched parts of sentences are considered as possible constituents. Nonterminals are assigned as they possibly generate the constituents. Such a nonterminal 'guess' in the assignments is called a *hypothesis*. For example, in the following two sentences, hypothesis is made for a nonterminal generates "*Oscar*", and other hypotheses for "*Cookie monster*", "*large, green*" and "*red*".

[*Oscar*] *sees the* [*large, green*] *apple*.

[*Cookie monster*] *sees the* [*red*] *apple*.

Next, the hypotheses are selected based on evaluation functions. The best function proposed, according to the experimental results in [Zaanen 2002a], is a probabilistic function which is called *leaf*. The function for a hypothesis *h* is noted $P_{leaf}(h)$, which is the number of all appearance of hypotheses that generate the same subsentence divided (normalized) by the

number of all appearance of all hypotheses. In other words, the evaluation function for a hypothesis is the number of brackets with the same subsentence into the total number of brackets in all the sentences.

The selected hypotheses are then used for creating grammatical rules, by assigning new symbols representing the subsentences, or called constituents, that are in pairs of brackets. Thus, the above example results in the following grammar rules:

$$S \rightarrow A \textit{ sees the } B \textit{ apple}$$

$$A \rightarrow \textit{Oscar}$$

$$A \rightarrow \textit{Cookie monster}$$

$$B \rightarrow \textit{large, green}$$

$$B \rightarrow \textit{red}$$

Then, the grammar rules extracted are returned by the framework as learning results.

There is some interesting work referring to Alignment Based Learning. Klein and Manning [Klein 2005, Klein 2001] first introduced a generative constituent-context model for unsupervised distributional induction of bracketed tree structures. Rather than context-free grammar, their model induces bracketing (the way to put brackets) of sentences. In the first phase of their method, a bracketing is chosen based on some distribution. In the second phase, the stochastic parameters are set using expectation-maximization (EM) algorithms.

An EM algorithm is an algorithm finding maximum likelihood estimations of stochastic parameters. The method has been proved to acquire a large amount of structure and requires no part-of-speech information (e.g. NP for noun phrase). However, the method still needs to work with word class data in additional to the input sentences.

A parallel-text is a text where sentences in different languages such as English and French are aligned. Kuhn [Kuhn 2004] proposed an indication method to use parallel-text for training combining with EM algorithms. By experiments, it is proved that this method results in better accuracy. This result is interesting. However it still needs alignment information between languages to work.

In Clark's work [Clark 2001], an algorithm is presented to learn grammar from tagged data. The algorithm uses distributional information of tag sequences to cluster the sequences. An entropy based criterion is used to select such clusters. It is proved that this algorithm is able to carry out unsupervised induction. However, according to the author, this work is preliminary in terms of computational complexity.

Bod [Bod 2006] estimated generalization of the all-subtrees Data-Oriented Parsing (DOP) algorithm to unsupervised parsing. In that algorithm, a tagged corpus is used as reference for parsing: on a given new sentence, the parsing tree is generated by selecting the most possible subtrees from all possible ones. According to their experiments, this proposed unsupervised

algorithm beat a widely used supervised model. Unfortunately, this method needs tags to work.

2.6 Discussions

In grammatical inference, the general tasks of learning languages are proved to be computational impossible. For regular languages, three classical learning methods were proposed by Angluin: learning regular sets from a representative sample and using membership queries [Angluin 1981], identifying regular sets using equivalence queries and membership queries [Angluin 1987b], and learning reversible languages [Angluin 1982]. Recently, state merging methods are proposed. However, all the approaches are to generate results as accepting a string or rejecting it. Though constructing separated automata for multiple language classes is a straight forward extension, the resulting models are constructed with unnecessarily space cost. Based on this observation, we propose in Chapter 3 to construct a single automaton representing multiple classes through learning from the given string samples.

Since reducing the size of automaton for multiple language classes is not applicable on all situations and constructing automata for a single language class is a more general case, we introduce cover automata on regular language learning in Chapter 4 to reduce the size of the learned model from regular languages. Cover automata [BalcRazar 1985, Dwork 1990, Shallit 1996] is a model designed to reduce complexity of finite language model. The idea of the cover automata for finite languages is to use a number limit to control the length of the words while using automata to control the structure of the strings. In this way, the number of

states in an automaton is reduced, for some information is given in the number controlling the length.

For stochastic inference, existing methods efficiently solve the problem of probabilistic parameter estimation. However, no efficient algorithm is available for identifying the structure of the learned stochastic model. For a classical stochastic model, Markov Chains model, in practice, there are many situations in which it is difficult to determine a fixed order for the model, as highly observed patterns may include sequences of different lengths at the same time. However, saving a probabilistic parameter for each possible sequential pattern is not efficient. Recurrent neural networks are found to be useful to learn sequential data in [Husken 2003]. However, it only learns sequential data without stochastic information. In Chapter 5, we present an approach for *recurrent neural network* (RNN) construction to identify mixed *k*-th order Markov Chains from given sample strings. By pruning based on statistical property captured through "purity value", the model size is reduced from being exponential.

In the existing context-free language learning methods, many of them need structural information, which means that the grammatical rules are partially given. Therefore, such methods are not practical since in real applications, there is usually no structural information available. While k-bounded CFG and SDL are found to be learnable, existing learning methods are based on structural queries, which are again not practical. The learning of ELL is efficient and query-free. However, the representing power of the language is very limited.

SEQUITUR is a very efficient application oriented method, but it is mainly for compression purpose as its resulting grammar can only generate the exact given samples with no generalization ability. Lattice and Squeeze methods creatively combined context-free grammar learning and regular language learning and generates interesting results. However, it depends on a given parse forest where the structural data are already given.

Alignment based learning shows interesting results: it needs no extra data other than the sample strings; it accepts input from real applications; it handles stochastic data. However, there are two problems remaining open in the alignment based learning framework [Zaanen 2002a]. One problem is that we need words in the two sentences having exact match; this hypothesis is too strong in applications. For example, suppose we have the following two sentences:

Book a trip to Sesame Street.

Show me Big Bird's house.

In the standard alignment based learning framework, no structure is found because there is no exact match of any words. If the algorithm has additional input to specify that the word "*Book*" and "*Show*" are of the same type, structure can be found as:

Book [*a trip to Sesame Street*].

Show [*me Big Bird's house*].

Another problem is that words in different types, if found in the same context, can be recognized as in the same type. As shown in the following example, the word "*biscuits*" and the word "*well*" are determined as of the same type.

Ernie eats [*biscuits*].

Ernie eats [*well*] .

These two problems of Alignment Based Learning reduced the precision of the algorithm output grammatical rules. To address these two problems, we proposed a framework called Profile Based Alignment Learning in Chapter 6, which reduces the number of invalid identified grammatical rules and improves the precision.

CHAPTER

3

Classification Automata and its Construction

In this chapter, we present a method to construct a single automaton for multiple language classes, called *classification automata*, from given string samples. We discuss algorithms to learn classification automata in different learning settings.

3.1 Notations

Let Σ be an alphabet. Let S be a finite set of strings in a language L over Σ. If π is a partition of S, then for any element $s \in S$ there is a unique element of π containing s, which we denote by $B(s,\pi)$ and call it as the block of π containing s. An acceptor (over Σ) is a quadruple $A=(Q, I, F, \delta)$ such that Q is a finite set of states, I and F are subsets of Q, and δ maps from $Q \times \Sigma$ to subsets of Q. Let A be a deterministic acceptor with initial state set I. Define the partition π_A by $B(w_1, \pi_A) = B(w_2, \pi_A)$ iff $\delta(I, w_1) = \delta(I, w_2)$. The prefix tree acceptor for a

given sample set S, is denoted as $PT(S)$. Let $A = (Q, I, F, \delta)$ be an acceptor, and let $L=L(A)$. The reverse of δ, denoted by δ^r (and corresponding acceptor, denoted by A^r) is defined by

$$\delta^r(q, a) = \{q' \mid q \in \delta(q', a)\}, \text{ for all } a \in \Sigma, q \in Q.$$

The acceptor A is said to be *0*-reversible iff both A and A^r are deterministic.

3.2 Constructing Classification Automata

3.2.1 Finite Automata Extension for Multiple Classes

To get classification automata, we consider an extended alphabet $\Sigma \cup \{\$_1, \$_2, \$_3, \ldots \$_n\}$. We append the string s_i a symbol called classification symbol $\$_j$, when s_i is in the j-th class. We use $\$(s_i)$ denote the classification symbol of string s_i. Formally, a sample string set S' for classification is defined as:

$S' = \{s'_i \mid s'_i = s_i \$(s_i), s_i \in S\}$

Suppose our sample strings, are:

s_1 = *abababaababababaaabbbababababa* in class 1

s_2 = *ababaabbbababababababaaaababa* in class 2

s_3 = *abababababababababa* in class 1

……

Then we modify the input strings by adding the classification symbol as their suffix. For the example above, we get:

s'_1 = *abababaababababaaabbbababababa*$\$_1$

s'_2 = *ababaabbbababababababaaaababa*$\$_2$

$s'_3 = abababababababababa\$_1$

……

We now construct automaton for *S'*. Let *A* be a deterministic acceptor with initial state set *I*. Define the classification state set of *A* as *CS(A)*, such that:

$CS(A) = \{ qc_i \mid qc_i \in Q$, there exists $\$_i$, $\delta(q, \$_i) \in F$, where *i* is the class number$\}$.

For all $qc_i \in CS(A)$, we denote qc_i as classification state in class *i*.

The acceptor *A* can be constructed by using our extended strings. We then make the following modification: omit the final state, mark all those classification states as new final states with their classification symbols. As a result, we get the classification automata, which can now be used for predicting the class for a new string. Its class will be the class of the symbol marked on the final state the automaton reaches in the end. Since we have no final state with more than one class attached, we can always get a string classified if the automaton accepts it. If not, the class of the string can be called as "unknown".

Suppose our resulting automaton using some learning method is like the one shown in Figure 3.1.

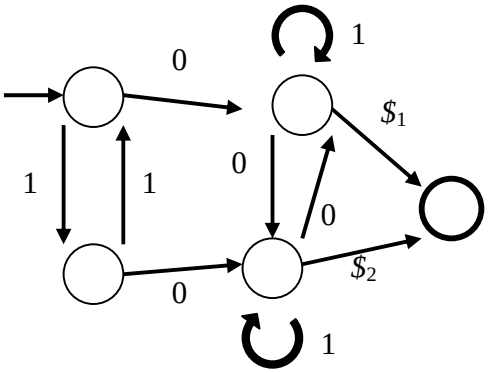


Figure 3.1 Resulting automaton

The classification automaton is constructed as follows.

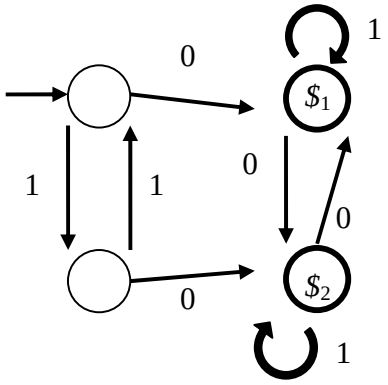


Figure 3.2 Classification automaton from the resulting automaton

Under this definition, for a method constructing classification automata, to prove its correctness, we need to prove the following two conditions.

(1) Every string *u* accepted by the resulting automaton should end with classification symbol, and contains only one classification symbol.

(2) If there is no conflict for the classification of the target concept, then there exists no string *u* such that the resulting automaton accepts $u\$_i$, and $u\$_j$, $i \neq j$, i, j are positive integers.

Suppose that after construction, some strings accepted by the automaton are as follows.

a. $01010010111\$_1$

b. $011\$_1100100$

c. $011\$_1100100\$_1$

d. $01010010111\$_2$

e. $01\$_1$

We say that, string *b*, *c* cannot satisfy condition 1 because string *b* does not end with classification symbol, and string *c* contains more than one classification symbol. String *a* and *d* cannot satisfy condition 2, for the same string is classified in two classes. In the following parts, we discuss the learning of classification automata on regular sets in different approaches.

3.2.2 Learning by Representative Sample and Membership Queries

By Algorithm 2.1 in Chapter 2, we have introduced the algorithm learning regular sets from a representative sample and using membership queries, algorithm ID [Angluin 1981]. Our idea for learning classification automata by representative sample and membership queries is to use membership queries to ensure our conditions 1 and 2. For each membership query to confirm an ending string, the only permitted ones are those that end with classification symbols. Formally, we have the following lemmas and theorems.

Lemma 3.1 In algorithm ID, if all string in the target concept ends with a single and unique classification symbol, the final states are never merged with non-final states.

PROOF Since all string ends with a single and unique classification symbol, any non-final state can only reach the final state by a non-empty string end with a classification symbol.

Meanwhile, the empty string λ is always a distinguishing string of the final states. Therefore, the Lemma follows.

Q.E.D.

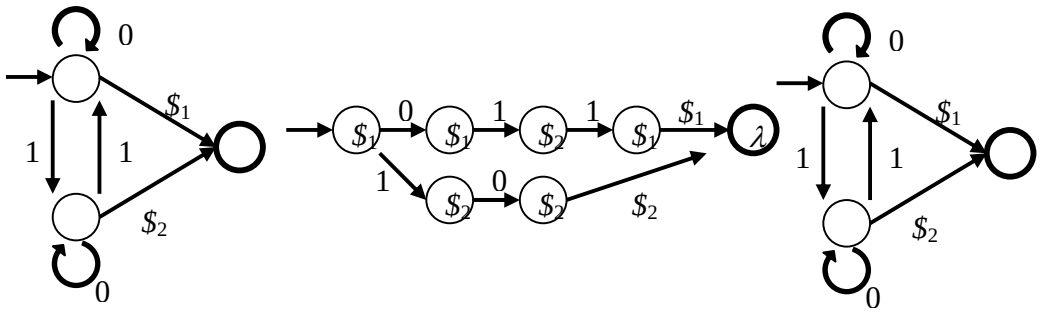
Theorem 3.1 Classification automata can be constructed from representative samples by membership queries.

PROOF For the first condition, since final states in the prefix tree acceptor only have in-arrows carrying classification symbol and the merge of such states will keep this feature. Hence, the resultant acceptor will only accept strings which end with, and contain only one classification symbol. Thus, condition 1 is satisfied. Suppose the second condition is false, we must have two states p, q, such that u accesses p; v accesses q, and $p\$_i$, $q\$_j$ ($i \neq j$) are both accepted. This implies that, for states p and q, they must have the ending string subset $\{\$_i, \$_j\}$. Hence, the answer to the membership queries about strings $u\$_i$, $u\$_j$, are both given "yes". This leads to contradiction. As condition 2 is also satisfied, the Theorem 3.1 follows.

Q.E.D.

With Theorem 3.1, we show that, by controlling the membership queries and representative samples, the classification automata can be constructed. Since the behaviors of the algorithms do not change with the extension symbols added, the complexity of the algorithm is not changed.

We use the representative sample set $S = \{011\$_1, 10\$_2\}$ to illustrate the training of classification automaton in this approach. The target automaton and the result are shown in Figure 3.3.



(a). target automaton (b) marking of states (c) resulting acceptor

Figure 3.3 A running example for the classification automaton by representative sample and membership queries

3.2.3 Learning by Equivalence Queries and Membership Queries

The details of the algorithm identifying regular sets using equivalence queries and membership queries in Dana Angluin's work [Angluin 1987b]. The idea is to use the counterexample strings returned by equivalence queries to maintain the tree of distinguishing strings, and thus find more states in the target automaton.

Our idea for learning classification automaton by equivalence queries and membership queries is to use equivalence queries to ensure our conditions 1 and 2. For each equivalence query that does not confirm with conditions 1 and 2, we just return the counter-examples. The proof is shown formally as follows.

Theorem 3.2 Classification automata can be constructed through equivalence queries and membership queries.

PROOF By our extension, the samples are attached classification symbols at the end of the strings. Then the sample strings can be used to answer queries. For equivalence queries, we can get the counter-examples as follows. If the result disagrees with condition 1, the oracle will return the counter example as a string does not end with classification symbol, or contains two different classification symbols. If the result disagrees with condition 2, the oracle will return the one not correctly classified in $u\$_i$ and $u\$_j$. For membership queries, the answer is true if the string is in the given sample, and the answer is false otherwise. Since, if there is any conflict with condition 1 or 2, an equivalence query will propose a counter example and the automaton will be adjusted until no more conflicts are found. In this way, algorithm Learn-Automaton will generate the resulting automaton, which must be a classification automaton.

Q.E.D.

With Theorem 3.2, we show that, returning counter-examples that do not agree with condition 1 or 2, classification automata can be constructed. Again, the complexity of the algorithm is not changed since the extension does not bring any changes to the running of the algorithm. Next, we train an example classification automaton for this approach using algorithm 2.2 (algorithm Learning Automaton [Kearns 1994]) on our model in Section 3.2.1.

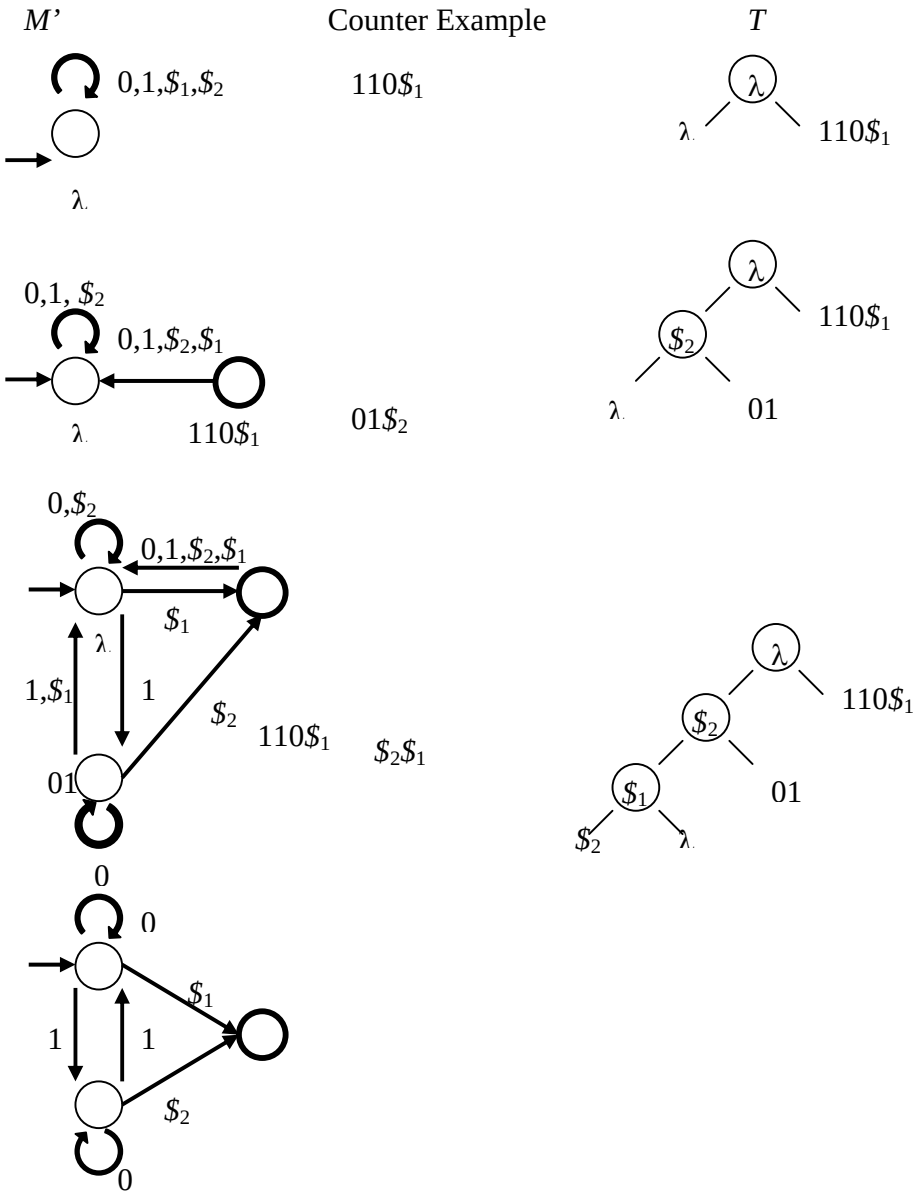


Figure 3.4 A running example for the Classification automaton by equivalence queries and membership queries

3.2.4 Learning Zero-Reversible Language

This extension is based on algorithm ZR [Angluin 1982]. The idea of this algorithm, called *ZR* (Algorithm 2.3). Since the algorithm *ZR* works with merging states, some of the states that are connected to different classification symbols may be merged in the resulting automata by applying algorithm *ZR*. This situation is illustrated in the figure given below: when a string reaches state A, then the classification automata does not know whether the string should be classified for class 1 or class 2.

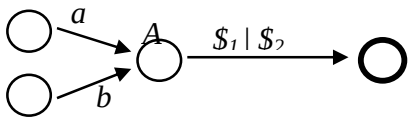


Figure 3.5 A string ends at state *A*, can not be classified

In other words, for the reversible language approach, the application of algorithm *ZR* on our extended strings may cause the merge of some states that are connected to different classification symbols. Our solution to this problem is to get the merging chain that causes the undesired merge. The algorithm ZR merges B_1 and B_2, if $\delta(B_1, a) = \delta(B_2, a) = B_3$ or $\delta^r(B_1, a) = \delta^r(B_2, a) = B_3$. So we find out the merge that forms the block B_3, and then find out this chain recursively. Then we split the two classification states by changing the labels into some subclass labels, such as $\$_{i,1}$ and $\$_{i,2}$.

The method we use for finding the corresponding merge is termed *find merge method*:

1. Find B_3 such that $\delta(B_1, a) = \delta(B_2, a) = B_3$ (or $\delta^r(B_1, a) = \delta^r(B_2, a) = B_3$).

2. Search the merging log, until we find which merge resulted in B_3.

3. Do 1 and 2 recursively, until $B_3 \in \{\text{final states of } A\}$.

4. Find all states with the same alphabet symbol pointing to B_3.

For instance, suppose we want to find the last merge that leads to $\delta(B_1, a) = \delta(B_2, a) = B_3$ (or $\delta^r(B_1, a) = \delta^r(B_2, a) = B_3$). We find out that B_3 was the merging result of some previous block pair B_1' and B_2' such that there exists B_3' and some $b \in \Sigma$ such that $\delta(B_1', b) = \delta(B_2', b) = B_3'$ (or $\delta^r(B_1', b) = \delta^r(B_2', b) = B_3'$), where $B_1' \cup B_2' = B_3$. Before the merge which forms B_3, the three situations are shown as follows:

(1) There are a transitions between B_1' and $\{B_1, B_2\}$, and there are a transitions between B_2' and $\{B_1, B_2\}$. Note that the transition direction could be both ways, but in the same merge, they should be in the same direction with the same symbol. Only one direction is shown in the figures.

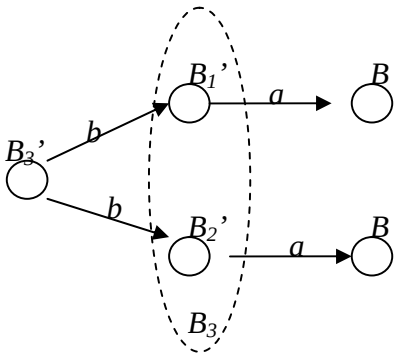


Figure 3.6 *Find merge* recursion situation 1

(2) There are *a* transitions between B_1' and $\{B_1, B_2\}$, but no *a* transition between B_2' and $\{B_1, B_2\}$.

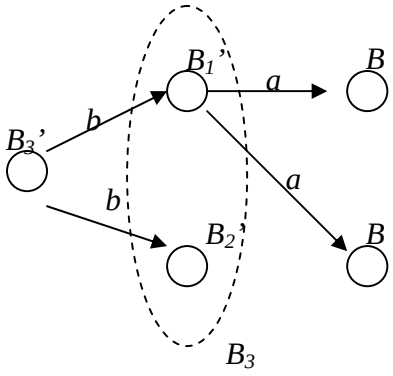


Figure 3.7 *Find merge* recursion situation 2

(3) Or, there are *a* transitions between B_2' and $\{B_1, B_2\}$, but no *a* transition between B_1' and $\{B_1, B_2\}$.

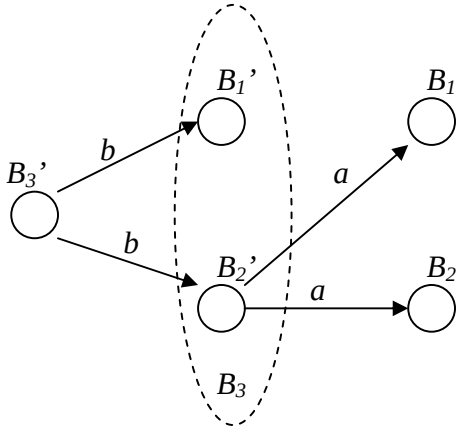


Figure 3.8 *Find merge* recursion situation 3

In the *find merge* method, if we get situation 1 (Figure 3.6), it means that the merge of B_1 and B_2 is caused by the merge of B_1' and B_2'. Recursively, next step is to find the merge which

cause the merge of B_1' and B_2'. Otherwise, if we get situation 2 (Figure 3.7), it means that the merge is not the one that causes the merge of B_1 and B_2. There must be another merge which leads to the merge of B_1 and B_2. Therefore, next step is to find an earlier merge which forms B_1'. Similar to situation 2, if we get situation 3 (Figure 3.8), next step is to find an earlier merge which forms B_2'.

The *split* method is to replace original classification symbols with new split classification symbols as $\$_{i,1}$ and $\$_{i,2}$. Then we run the algorithm again, with the changed symbols. We call our algorithm to solve this problem *cl-ZR*.

Algorithm 3.1 Algorithm *cl-ZR*:

//Initialization
- Construct the prefix tree PT(S)of the given sample S.
- Acceptor A=PT(S)

//Loop
- For all states
- Find B_1 and B_2 such that $\delta(B_1, a) = \delta(B_2, a) = B_3$
- If no such B_1 and B_2 exist, output and exit.

    --If B_1 and B_2 contain different classification states

        -*split(find merge*(B_1 ,B_2))

        -*restart* loop

    --Else merge B_1 and B_2

//End of algorithm

We prove the correctness of *cl-ZR* as follows.

Lemma 3.2 In any stage of *cl-ZR*, for any final state block B_1, non-final state block B_2, and any block B_3, the following are never satisfied:

(*i*) $\delta(B_1, a) = \delta(B_2, a) = B_3$

(*ii*) $\delta^r(B_1, a) = \delta^r(B_2, a) = B_3$.

PROOF At the first stage of the algorithm, the prefix tree is constructed, the final states are all merged as one final state, and we denote this only one final state as q_f. State q_f has no out-arrows. All in-arrows to q_f carry a classification symbol. We are going to prove that no further merge will happen on q_f. Suppose that we have $B_1 = \{q_f\}$, for a certain a in alphabet, there exist B_2 and B_3 satisfying $\delta(B_1, a) = \delta(B_2, a) = B_3$. That is $\delta(\{ q_f \}, a) = \delta(, a) = B_3$, for some $q_i \in B_2$. Note that there is no $\delta(\{ q_f \}, a)$ at all, that lead to a contradiction. On the other hand, suppose that we have $B_1 = \{q_f\}$, for a certain a in alphabet, there exist B_2 and B_3 satisfying $\delta^r(B_1, a) = \delta^r(B_2, a) = B_3$. That is $\delta^r(\{ q_f \}, a) = \delta^r(, a) = B_3$, $q_i \in B_2$. However, for all $\delta^r(\{ q_f \}, a)$, a is classification symbol, and for all $\delta^r(, a)$, a is a non classification symbol. Hence, it leads to a contradiction.

Q.E.D.

The result in Lemma 3.2 can also be summarized in other words in the form of the following theorem.

Theorem 3.3 In the algorithm *cl-ZR* (Algorithm 3.1), final state and non-final state are never merged.

Lemma 3.3 After the merge of the final states, all blocks B_1 and B_2 satisfying $\delta(B_1, a) = \delta(B_2, a) = B_3$ are classification states, and no such blocks B_1 and B_2 satisfying $\delta^r(B_1, a) = \delta^r(B_2, a) = B_3$ exist.

PROOF Before the merge of the final states, we have the prefix tree of the given sample. In this stage, no block contains more than one state except the final state block. Since we have Theorem 3.3, we only need to discuss the non-final states here. The last symbol of every sample string is a classification symbol. After the merge, only the final states are merged as one. We still denote it as q_f. It is obvious that the prefix tree is deterministic, and for every state in the prefix tree, it has at most one parent. That is, for all q in Q_0, if there exist $q_1 \in Q_0$, $q_2 \in Q_0$, such that for a certain a in alphabet, $\delta(q_1, a) = \delta(q_2, a) = q$, we have $q_1 = q_2$. Furthermore, after the final state merge, the δ_0 does not change at all, except for those containing final states. So it is easy to say that, all states which do not appear together with final states will remain dual-direction deterministic. From the proof of Lemma 3.1, it follows that, those states contained in the δ_0 are exactly the members of the classification state set. Hence, this Lemma follows.

Q.E.D.

Theorem 3.4 For any non-final state blocks B_1 and B_2, the first merge found by the *find merge method* must be a merge of two classification symbols with the same class.

PROOF In *cl*-ZR, the merges are done in a sequential order. Some merges must be

done after other merges, as the later merges need the result of earlier merges. Suppose we have $\delta(B_1, a) = \delta(B_2, a) = B_3$, and it is the first merge; hence B_3 must be a block which contains a single state; let this state be q_3. That is, we have $\delta(q_1, a) = \delta(q_2, a) = q_3$, $B_3 = \{q_3\}$, $q_1 \in B_1$, $q_2 \in B_2$. If q_1, q_2 are not classification states and q_3 is not the merged final state, then it will lead to a contradiction to Lemma 3.2. Hence Theorem 3.4 follows.

Q.E.D.

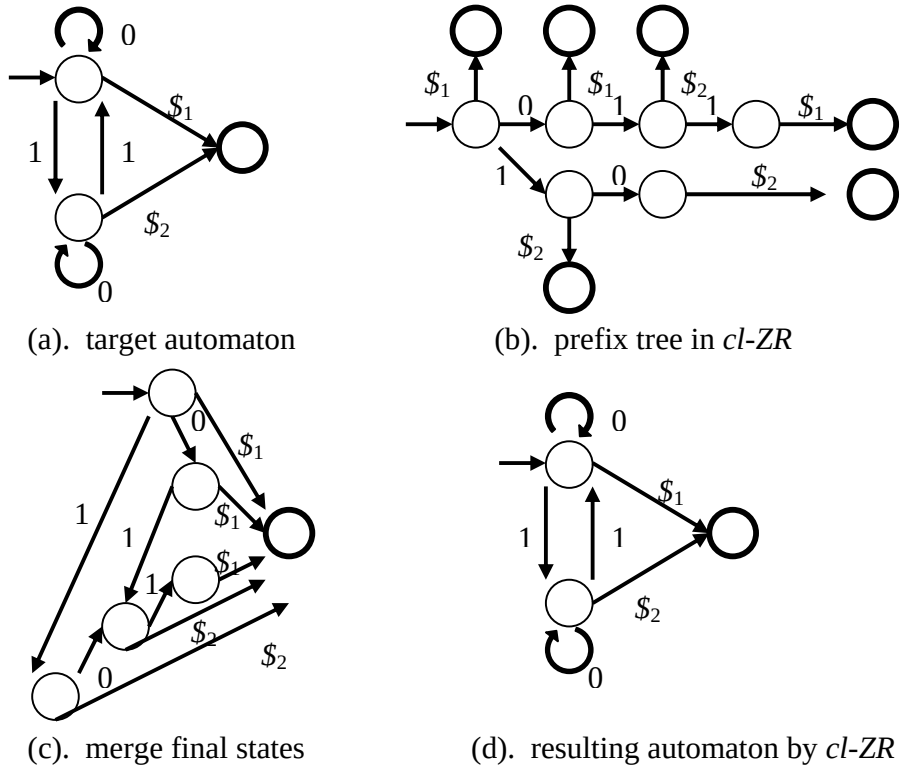
Theorem 3.5 The algorithm *cl-ZR* needs $O(mn\alpha(n))$ operations, where m is the number of strings, n is one more than the sum of the lengths of the input strings and α is a very slowly growing function (α is defined in [Tarjan 1975]).

PROOF In *cl-ZR*, the worst case is that we run *ZR* on the sample set once for every split, and we have every string split as a subclass. Hence, we need to run *ZR* for every input string. Thus, in our algorithm, computations as needed in Angulin's *ZR* (which needs $O(n\alpha(n))$ operations) are performed m times. Hence, the running time of *cl-ZR* is bounded by $O(mn\alpha(n))$.

Q.E.D.

For the target automaton, suppose we have the training sample $S = \{\$_1, 0\$_1, 01\$_2, 011\$_1, 1\$_2, 10\$_2,\}$. We use this sample set on *cl-ZR*. Firstly, *cl-ZR* constructs the prefix for the

given sample *S*, as showed in Figure 3.9 (b). Then it merges the final states. (Figure 3.9(c)) After steps of operations, the resulting automaton is constructed. (Figure 3.9 (d))



(a). target automaton (b). prefix tree in *cl-ZR*

(c). merge final states (d). resulting automaton by *cl-ZR*

Figure 3.9 Target automaton and running of *cl*-ZR

3.3 Using Negative Examples on Classification Automata

We take the negative examples as a separate class. Algorithm 3.1, Algorithm *cl-ZR*, is implemented and tested on both positive and negative samples.

Formally, we have:

The alphabet for taking input as positive and negative samples is extended as $\Sigma \cup \{\$_+, \$_-\}$. We append an input string s_i a symbol $\$(s_i) \in \{\$_+, \$_-\}$, denoting the class of string s_i, $\$_+$ for positive and $\$_-$ for negative. Hence, a sample string set *S'* for classification is defined as:

$S' = \{s'_i \mid s'_i = s_i \, \$(s_i)$, where $\$(s_i) \in \{\$_+, \$_-\}$ is the classification symbol of s_i , $s_i \in S\}$

By limiting the classification symbols to positive and negative class, we can construct automata that can learn information from positive sample and negative sample at the same time.

Therefore, when the resulting automaton is found, it can naturally separate the negative class and the positive class as different classes. Moreover, we can have the automata constructed incrementally, so that, for each input string we take, whether it is positive or negative, the construction of the multi-classifier will get a clear separation between the positive class and the negative one.

3.4 Experimental Results

Our algorithms were applied on house renting data in the exchange system of Nanyang Technological University. We grouped sentences by the information categories: namely, rooms in the flat, furnishing condition and extra information. Some example sentences describing number of rooms are given below:

A 4+1 flat is for rent.

Three + one flat for NTU students!

A three bedroom flat is for rent.

Each group of sentences is further divided into classes by the contents of the sentences. For example, sentences that with two-bedroom flats are class one and those with three-bedrooms are class two, and so on. We trained our classification automata to learn such string classes and then evaluate the result by using it to classify unseen strings. For each category, we use 1000 sentences in our data, and taking 150, 200, 250, 300, 350, 400 and 450 of them as training set. The experimental results are shown below. Only results for *cl*-ZR are shown, because experiments on the other two approaches are not applicable due to their requirements for query.

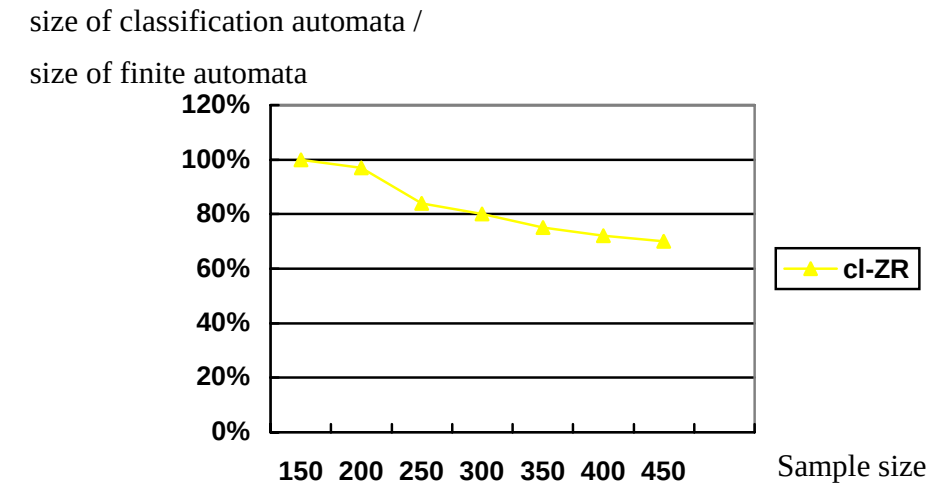


Figure 3.10 size of classification automata / size of finite automata by ZR

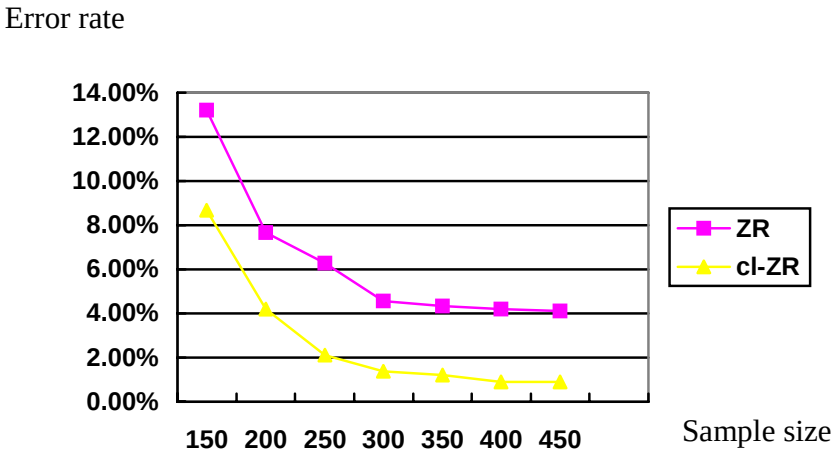


Figure 3.11 Error rates of *cl*-ZR and ZR

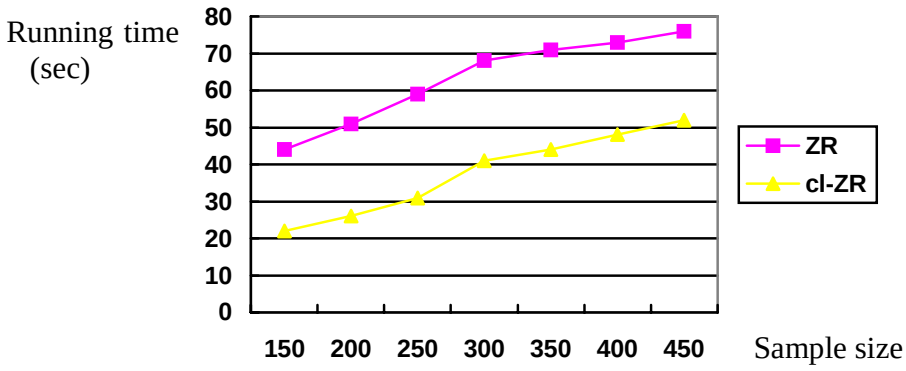


Figure 3.12 Running time (in sec) of *cl*-ZR and ZR

For each data set, one classification automaton is constructed using cl-ZR, and multiple automata are constructed for each class of ZR. We use the nodes of the automata as a measurement for model size. Figure 3.10 shows the average value of the sizes of the classification automata divided by the total sizes of the automata by ZR. When sample size is small, classification automata is almost in the same size of separated automata. With more samples available, the size is reduced up to 30%. This is because more nodes are shared by putting different classes together with more samples.

Figure 3.11 illustrates the predicting error rates of ZR and *cl*-ZR. *cl*-ZR gets lower error rates because of two reasons:

(1) When sample number is relatively small, samples in different classes may help each other to generalize.

(2) When sample number grows larger, the splitting method in *cl*-ZR prevents overgeneralization.

Figure 3.12 shows the running time of the algorithms ZR and *cl*-ZR. It can be seen in the chart that our algorithm *cl*-ZR runs in less time than algorithm ZR.

3.5 Conclusions

We proposed an extension of automata, classification automata, to reduce the learned model size. We also proposed the extension on three classical regular language learning methods by Angluin and designed algorithms to construct classification automata for 0-reversible languages, by state merging and splitting according to backtrack information identified in three conditions. The correctness and efficiency of the algorithms has been proved in the form of theorems. Experiments show that our method can save up to 30% model size. However, this method is only useful for multiple language cases. When samples from only one language class are given, classification automata degrade to normal automata. To reduce the model size in single class language learning, we introduce cover automata in Chapter 4.

CHAPTER

4

Construction of Minimal Cover Automata

Cover automata are used as learning model to reduce the model size. In some applications, we have strings from L instead of having deterministic finite automaton of L. Since a cover automaton accepts exactly the language L when the length is limited and will accept more strings for longer word length, the cover automata may be used in inferring regular languages. We give an efficient algorithm that constructs minimal cover automata from positive sample set of finite languages.

4.1 Notations

The alphabet is a fixed finite nonempty set Σ of symbols. Σ^* denotes the set of all finite strings over Σ. λ denotes the *empty string*. The length of the string w is denoted by $|w|$. Let $\Sigma^l = \{w \mid |w|=l\}$, and $\Sigma^{\leq l} = \{w \mid |w| \leq l\}$. The concatenation of the strings u and v is denoted by uv. The string u is a *prefix* of the string v iff there exists a string w such that $uw = v$. A *language* L is a subset of Σ^*. A *finite language* L is a subset of $\Sigma^{\leq l}$, where l is the length of the longest string in L. A finite language L can also be called a *positive sample set*, since L is a finite set of string(s). Define $\mathrm{Pre}(L)=\{u \mid u$ is a prefix of $w, w \in L\}$ and $\mathrm{Pre}(w,i)=\{u \mid uv=w, |u|= i\}$.

4.2 Construction of prefix tree acceptor

To construct a prefix tree acceptor from finite language L, we scan each string in L from the beginning to the end, constructing transitions which are still not a part of the automaton. After scanning all the strings in L, the prefix acceptor is constructed. For the finite language L in Example 4.1, we illustrate the construction of the prefix tree acceptor in Figure 4.1.

Example 4.1 Let $L=\{a, aa, aaa, baa\}$, PT(L) is shown in Figure 4.1.

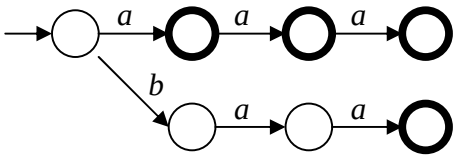
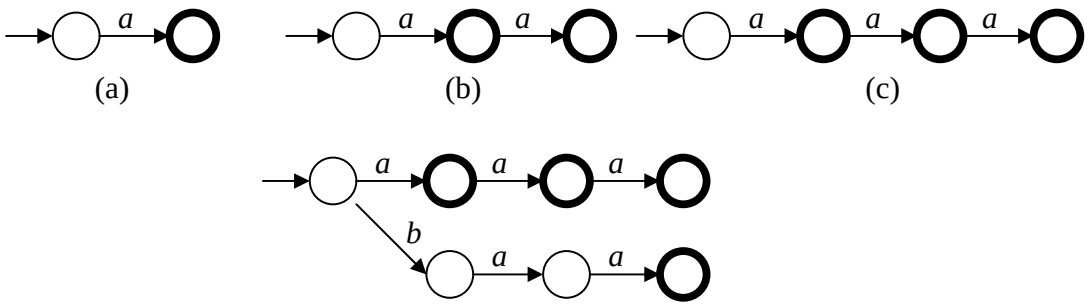


Figure 4.1 Prefix tree acceptor for *L*

]



(a) (b) (c)

(d)

Figure 4.2 A running example of constructing prefix acceptor from language *L*={*a*, *aa*, *aaa*, *baa*}. (a), (b), (c), (d) are the results after taking string *a*, *aa*, *aaa*,*baa*

Formally, the algorithm is as follows.

Algorithm 4.1 *Get*PT*(L)*

```
Let Σ be the alphabet,
q0=φ, F=φ, δ=φ, Q =φ .
if L=φ , output PT(L) =(Σ, Q, q0, δ, F)
else{   q0={λ}
        for( w∈L ){
                for i=1 to |w|{
                        if Pr(w,i) ∉Q{
                                Q=Q ∪{ Pr(w,i)}
                                a=the ith letter in w
                                δ=δ ∪{δ( Pr(w,i-1),a)= Pr(w,i) }
                                }
                        }
                F=F ∪{w}
                }
    }
Output PT(L) =(Σ, Q, q0, δ, F)
```

By definition, for each $p \in Q$, there is a unique string u such that $\delta(q_0, u)=p$. Therefore, in a prefix tree acceptor, when we use a state p it represents the string $\delta(q_0, p)=p$ at the same time.

Example 4.2 We still take the finite language L in Example 4.1, and the prefix tree acceptor we constructed for L. We can say that, for each state there is a corresponding string that 'accesses' the state from the start state. (Illustrated in Figure 4.3)

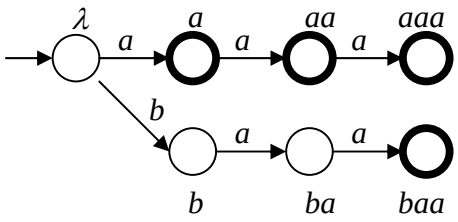


Figure 4.3 States in prefix tree acceptor for *L* and their unique strings

4.3 Construction of the Maximal Dissimilar Set

4.3.1 Deterministic Finite Cover Automata (DFCA)

Definition 4.1 [Campeanu 2001] A deterministic finite automaton is a quintuple $A=(\Sigma, Q, q_0, \delta, F)$, where Σ is the alphabet and Q are finite nonempty set of states, $q_0 \in Q$ is the starting state, $F \subseteq Q$ is the set of final state(s), and δ: $Q \times \Sigma \rightarrow Q$ is the transition function. δ can be extended from $Q \times \Sigma$ to $Q \times \Sigma^*$ by $\delta(q, \lambda) = q$ and $\delta(q, aw) = \delta(\delta(q, a), w)$, $q \in Q$, $a \in Q \times \Sigma$, $w \in \Sigma^*$. Denote $\#Q$ the cardinality of Q. The language recognized by the automaton A is $L(A)=\{w \in \Sigma^* | \delta(q_0, w) \in F\}$. Let l be the length of the longest word(s) in the finite language L. A DFA A such that $L(A) \cap \Sigma^{\leq l} = L$ is called a *deterministic finite cover automaton* (DFCA) of L.

Remark 4.1 By definition, a prefix tree acceptor PT(L) =(Σ, Q, q_0, δ, F) is a DFCA of L and such that is a DFA of L.

4.3.2 L-similarity

By definition, DFCA of L accepts finite language L when word length is limited by l. In other words, within the scope of $\Sigma^{\leq l}$, DFCA is consistent with language L. In fact, similar concept also applies to strings in L and states in *DFA*. L-similarity is used to describe such a concept, which is introduced in [Kaneps 1990]. If two strings or states have the same behavior within the length limitation of L, then they are considered to have L-similarity. Formal definition is described below.

Definition 4.2 [Campeanu 2001] Let Σ be an alphabet, $L \subseteq \Sigma^*$ a finite language, and l the length of the longest word(s) in L. Let $x, y \in \Sigma^*$. Define the following relations:

(1) $x \sim_L y$ if for all $z \in \Sigma^*$ such that $|xz| \leq l$ and $|yz| \leq l$, $xz \in L$ iff $yz \in L$;

(2) $x \not\sim_L y$ if $x \sim_L y$ does not hold.

The relation $\sim_L$ is called similarity relation with respect to L. If the language L is obvious in context, then we will omit L and denote $\sim$ and $\not\sim$.

For example, in the strings of example 4.1 and its prefix tree acceptor, the two strings a and aa (or the two states represented by the strings in the prefix tree acceptor) have L-similarity, because within the length limitation 3, the only strings which can be added to the end of them are λ, a and b. Since $a\lambda \in L$, $aa\lambda \in L$, $aa \in L$, $aaa \in L$, $ab \notin L$, $aab \notin L$,

i.e., *a* and *aa* have the same behavior on *L*, and so that they are considered to have *L*-similarity, say $a \sim aa$.

4.3.3 Minimal DFCA

Definition 4.3 Let $A=(\Sigma, Q, q_0, \delta, F)$ be a DFCA of a finite language *L*. We say that *A* is a *minimal* DFCA of *L* if for every DFCA $B = (\Sigma, Q', q_0, \delta', F')$ of *L* we have $\#Q \leq \#Q'$.

To get a minimal DFCA, the same idea of minimizing DFA is used. States with the same behavior will be merged as one state. Related definitions are given as follows.

Definition 4.4 [Campeanu 2001] Let *L* be a finite language.

(1) A set $S \subseteq \Sigma^*$ is called an *L*-similarity set if $x \sim_L y$ for every pair $x, y \in S$.

(2) A sequence of words $[x_1,\ldots, x_n]$ over Σ is called a dissimilar sequence of *L* if $x_i \not\sim_L x_j$ for each pair i, j, $1 \leq i,j \leq n$ and $i \neq j$.

(3) A dissimilar sequence $[x_1,\ldots, x_n]$ is called a canonical dissimilar sequence of *L* if there exists a partition $\pi = \{S_1,\ldots, S_n\}$ of Σ^* such that for each i, $1 \leq i \leq n$, $x_i \in S_i$, and S_i is an *L*-similarity set.

(4) A dissimilar sequence $[x_1,\ldots, x_n]$ of *L* is called a maximal dissimilar sequence of *L* if for any dissimilar sequence $[y_1,\ldots, y_m]$ of *L*, $m \leq n$.

The relationship between above concepts is proved in [Campeanu 2001]. The theorems and corollaries related to our work are listed below.

Theorem 4.1 A dissimilar sequence of L is a canonical dissimilar sequence of L if and only if it is a maximal dissimilar sequence of L.

Corollary 4.1 For each finite language L, there is a unique number N(L) which is the number of elements in any canonical dissimilar sequence of L.

Theorem 4.2 Any minimal DFCA of L has exactly N(L) states.

It is shown by the above results that a minimal DFCA can be constructed by finding a canonical dissimilar sequence and the corresponding partition of Σ^*. Put in the context of DFA, it is to find a sequence of states, where each state represents a set of states with the same behavior with respect to L.

4.3.4 Strings Not in Prefix Tree Acceptor

In the definition of prefix tree acceptor, dead state is not defined. However, in the definition of L-similarity, there sometimes exists such a string u that $u \in \Sigma^*$, but u is not a prefix of L. To apply the L-similarity relation to a prefix tree acceptor, it is needed to define those strings not represented by any state in a prefix tree acceptor.

Lemma 4.1 and 4.2 show that all strings that do not have a corresponding state have L-similarity relation with each other.

Lemma 4.1 Let PT(L) =(Σ, Q, q_0, δ, F) be a prefix tree acceptor of L, $u \in \Sigma^*$, $u \notin Q$ iff for any $v \in \Sigma^*$ such that $|uv| \leq l$, $uv \notin L$.

PROOF (1). $u \notin Q$,*i.e.*, $u \notin$ Pre(L), *i.e.*, for any string $v \in \Sigma^*$, that $uv \notin L$. (2) Suppose that for any $v \in \Sigma^*$ such that $|uv| \leq l$, $uv \notin L$. Consider $v' \in \Sigma^*$ such that $|uv| > l$, since l is the length of longest word(s) in L, $uv \notin L$. Hence, for any $v \in \Sigma^*$, $uv \notin L$. $u \notin$ Pre(L)=Q. From (1) and (2) the Lemma holds.

Q.E.D.

Lemma 4.2 Let PT(L) =(Σ, Q, q_0, δ, F) be a prefix tree acceptor of L, $p,q \in \Sigma^*$, if $p \notin Q$, $q \notin Q$, then $p \sim q$.

PROOF From Lemma 4.1, $p \notin Q$, $q \notin Q$ means that, for any $v \in \Sigma^*$ such that $|pv| \leq l$, $pv \notin L$, and for any $v' \in \Sigma^*$ such that $|qv'| \leq l$, $qv' \notin L$. Therefore, $p \sim q$.

Q.E.D.

Since all strings not in the state set of Pre(L) are in the same L-similarity set, the L-similarity defined on Σ^* is exactly the same on the elements of Pre(L) and its complement, and so that when we say $p \sim q$, it means string p has L-similarity relation with string q, and state p has L-similarity relation with state q.

For construction of minimal cover automata from sample strings, we have the following theorem.

Theorem 4.3 Let PT(L) =(Σ, Q, q_0, δ, F) be a prefix tree acceptor of L. If a sequence of words $[p_1,\ldots, p_n]\in Q$ such that p_i !~ p_j for each pair $i\neq j$, $1\leq i,j \leq n$, and there exists a partition $\pi=\{S_1,\ldots, S_n\}$ of L such that for each i, $1\leq i\leq n$, $p_i\in S_i$, and S_i is an L-similarity set, then the sequence of words $[p_1,\ldots, p_n, u]$ is a canonical dissimilar sequence of L, where u is any string such that $|u|=l+1$.

PROOF Let $S^-=\{w|\ w\notin L\ \}$, from lemma 4.2 we know that S^- is an L-similarity set, and $u\in S^-$. $S^-\cup L=\Sigma^*$ and $S^-\cap L=\phi$. So we have a partition $\pi=\{S_1,\ldots, S_n, S^-\}$ of Σ^* such that for the sequence of words $[p_1,\ldots, p_n, u]$, for each i,$1\leq i\leq n$, $p_i\in S_i$, S_i is an L-similarity set, $u\in S^-$ and S^- is an L-similarity set. Therefore the theorem holds.

Q.E.D.

By the construction described in Theorem 4.3, if we can find a sequence of words $[p_1,\ldots, p_n]\in Q$ such that p_i !~ p_j for each pair $i\neq j$, $1\leq i,j \leq n$, and there exists a partition $\pi=\{S_1,\ldots, S_n\}$ of L such that for each i, $1\leq i\leq n$, $p_i\in S_i$, and S_i is an L-similarity set, we will get a canonical dissimilar sequence of L. We denote such a $[p_1,\ldots, p_n]\in Q$ a *live canonical dissimilar sequence* of L.

4.3.5 Finding a Live Canonical Dissimilar Sequence

In this section, we present a method which constructs a live canonical dissimilar sequence of L and its corresponding L-similarity sets from the PT(L), *i.e.*, a dissimilar sequence $[p_1,\ldots, p_n]$ and the sets $\{S_1,\ldots, S_n\}$ on Σ such that for each i, $1\le i\le n$, $x_i\in S_i$, and S_i is an L-similarity set. Then, if we construct a DFCA whose state set is such a $\{S_1,\ldots, S_n\}$, by Theorem 4.1, Corollary 4.1, and Theorem 4.2, the acceptor is a minimal DFCA (with the dead state omitted).

By definition, such a set can be found by comparing each pair of states for all possible w of Σ^*, but this will lead to exponential complexity. To solve this problem, we will use an L-similarity recursion rule in the context of prefix tree.

Definition 4.5 Let $A=(\Sigma, Q, q_0, \delta, F)$ be a DFA. For each state $q\in Q$, level(q)= $\min\{|w| \mid \delta(q_0, w)=q\}$, *i.e.*, level($q$) is the length of the shortest path from the initial state to q. Denote $D_{-1}(A) = \{s\in Q|\delta(s,w)\notin F$, for all $w\in\Sigma^*\}$; for each $s\in Q$ let $\gamma_s(A)=\min\{w|\delta(s,w)\in F\}$, and $D_i(A)=\{s\in Q| \ |\gamma_s(A)|=i\}$, for each i=0, 1, ... Denote $d_A(s) =|\gamma_s(A)|$, *i.e.*, $d_A(q)$ is the length of the shortest path from q to the nearest final state.

If automaton A is obvious in context, then we use D_i, γ_s and d(s) instead of $D_i(A)$, $\gamma_s(A)$ and $d_A(s)$.

Example 4.3 Take the prefix tree acceptor in Example 4.1, suppose its states are as marked in Figure 4.4.

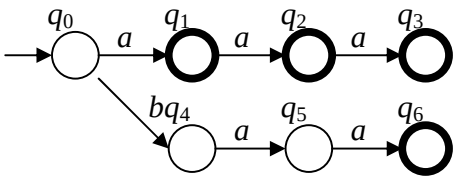


Figure 4.4 Marked states in PT(*L*)

We have level(q_0)=0, level(q_1)=1, level(q_2)=2, level(q_3)=3, level(q_4)=1, level(q_5)=2, level(q_6)=3, and d(q_0)=1, d(q_1)=0, d(q_2)=0, d(q_3)=0, d(q_4)=2, d(q_5)=1, d(q_6)=0, so that, D_0={q_1, q_2, q_3, q_6}, D_1={q_0, q_5 } and D_2={q_4}.

Lemma 4.3 Let PT(L) =(Σ, Q, q_0, δ, F) be a prefix tree acceptor of a finite language L. $p \in Q$, $a \in \Sigma$, $\delta(p, a) \in Q$ then level($\delta(p, a)$)=level(p)+1.

PROOF $\delta(p, a)=\delta(q_0, pa) \in Q$. Since in prefix tree acceptor, pa is the only string such that $\delta(q_0, pa)=pa$. $|pa|=p+1$, then level($\delta(p, a)$)=level(p)+1.

Q.E.D.

Lemma 4.4 Let PT(L) =(Σ, Q, q_0, δ, F) be a prefix tree acceptor of a finite language L. For p, $q \in Q$, level(p)<l, level(q)<l, if for all $a \in \Sigma$, $\delta(p, a) \sim \delta(q,a)$, then $p \sim q$.

PROOF Suppose $p!{\sim}q$, and suppose i = level(p)≥ level(q), there must be a symbol $a\in\Sigma$ and a string $w\in\Sigma^{\leq l-i-1}$ such that $\delta(p, aw) \in F$ and $\delta(q, aw) \notin F$, or $\delta(q,aw) \in F$ and $\delta(p,aw)\notin F$. Hence, we have $\delta(\delta(p, a),w)\in F$ and $\delta(\delta(q, a),w)\notin F$, or $\delta(\delta(q,a),w)\in F$ and $\delta(\delta(p,a),w)\notin F$. By lemma 4.3, we also have level($\delta(p, a)$) = level(p)+1 = i+1≥ level($\delta(q,a)$) = level(q)+1. That means $\delta(p, a)!{\sim}\ \delta(q,a)$. However, for all $a\in\Sigma$, $\delta(p, a){\sim}\ \delta(q,a)$. This is a contradiction.

Q.E.D.

Lemma 4.5 Let PT(L) =(Σ, Q, q_0, δ, F) be a prefix tree acceptor of a finite language L. For p, $q\in Q$, level(p)<l, level(q)<l, if there exists $a\in\Sigma$, $\delta(p, a)!{\sim}\ \delta(q,a)$, then $p!{\sim}q$.

PROOF Suppose i = level($\delta(p, a)$)≥ level($\delta(q, a)$). There must be a string $w\in\Sigma^{\leq l-i}$ such that $\delta(\delta(p, a),w) \in F$ and $\delta(\delta(q, a),w) \notin F$, or $\delta(\delta(q,a),w) \in F$ and $\delta(\delta(p,a),w)\notin F$. Hence, we have the string $aw\in\Sigma^{\leq l-i+1}$, such that $\delta(p, aw) \in F$ and $\delta(q, aw) \notin F$, or $\delta(q,aw) \in F$ and $\delta(p,aw)\notin F$. Since level(p)=level($\delta(p, a)$)-1= i-1≥ level(q)=level($\delta(q, a)$)-1, $p!{\sim}q$. Q.E.D.

Lemma 4.4 and 4.5 show that if we know the similarity between all possible $\delta(p, a)$ and $\delta(q,a)$, we can know the L-similarity relation between p and q. The following result is proved in [Campeanu 2001], which states that for states with different d(p) they must be L-dissimilar to each other.

Lemma 4.6 Let $A=(\Sigma, Q, q_0, \delta, F)$ be a DFCA of a finite language L, and $p \in D_i$; $q \in D_j$. If $i \neq j$, $i, j \geq 0$ then $p \not\sim_A q$.

By Lemma 4.6, we only need to calculate the similarity when d(p)=d(q). Then we need to initialize the recursion. If one state in a pair of states has level that equals the maximum length l of a finite language L, then Lemma 4.4 and 4.5 is not applicable. To tackle this situation, we introduce Lemma 4.7 below. In this situation, the states with maximum level cannot be attached any string with length more than zero. Again, we take Example 4.3. If any other level-0 state is compared to q_3, they will be considered L-similar, because they all have $\{\lambda\}$ to end, and within the limitation of the maximum length, no other string should be considered. We state this result formally in Lemma 4.7 below.

Lemma 4.7 Let PT(L)=($\Sigma, Q, q_0, \delta, F$) be a prefix tree acceptor of finite language L, d(p)=d(q), if level(p) = l and/or level(q) = l, $p \sim q$.

PROOF If level(p) = l and/or level(q) = l, then at least one of p and q is one of the longest words in L. Suppose p is a longest word, $\delta(p, \lambda) \in F$, d(p)= d(q)=0, $\delta(q, \lambda) \in F$. So we have for all $w \in \Sigma^{\leq l-l=0}=\{\lambda\}$, $\delta(p, \lambda) \in F$ and $\delta(q, \lambda) \in F$. Hence the lemma follows. Q.E.D.

Now we are ready to present the algorithm. In the algorithm, we calculate the similarity from high level to low level, for each D_i, and create a list_i to save all the states which have been scanned in D_i. For each q in the list_i, check then the similarity

between q and p, if $p \sim q$, then q is put in the similarity state set S_p. Note that while processing p and q, level(p)<l, level(q)<l, (which means Lemma 4.4 and 4.5 should be used to calculate, not Lemma 4.7), all states which have higher level than p have been scanned. Because q is in the list_i, level(q)≥level(p), so that level($\delta(p,a)$)>level(p) and level($\delta(q,a)$)>level(p). Since both $\delta(p,a)$ and $\delta(q,a)$ have been scanned, their similarity results are available from the sets $S_{\delta(p,a)}$ and $S_{\delta(q,a)}$.

Algorithm 4.2 Similarity_Set(PT(L))

```
for all p∈Q calculate level(p) and d(p)
for all p∈Q, according to d(p), calculate D_i(PT(L))
initialize list_i = φ for each D_i
initialize S_p = φ for each p
for p∈Q from level l to level 0 {
        for q in the list_i {
                if max(level(p), level(q))=l then put q in S_p //p~q, lemma 4.7
                else
                    for a∈Σ{
                        if δ(p,a)!~ δ(q,a), then p!~q //lemma 4.5
                        }
                    if for all a∈Σ, {
                        δ(p,a)~ δ(q,a), put q in S_p //p~q, lemma 4.4
                    }
        }
        put p in the list_i
}
```

The steps of Algorithm 4.2 are illustrated in Example 4.4 below.

Example 4.4 A running example is given on the prefix tree acceptor that we get from Section 4.2.

$D_0=\{q_1, q_2, q_3, q_6\}$,
$D_1=\{q_0, q_5\}$,
$D_2=\{q_4\}$.

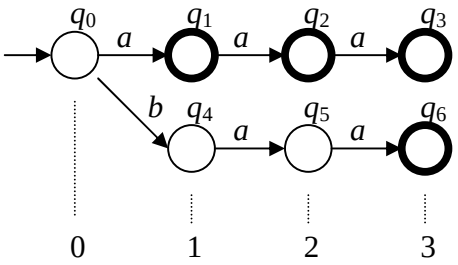


Figure 4.5 Levels of states in prefix tree acceptor

State processing	$list_i$	L-similar states
q_3	$list_0$ {}	
q_6	$list_0$ $\{q_3\}$	q_3
q_2	$list_0\{q_3, q_6\}$	q_3, q_6
q_5	$list_1\{\}$	
q_1	$list_0\{q_3, q_6, q_2\}$	q_3, q_6, q_2
q_4	$list_2\{\}$	
q_0	$list_1\{q_5\}$	q_5

Figure 4.6 The running of finding *L*-similarity sets

So the live canonical dissimilar sequence is $[q_1, q_0, q_4]$. The corresponding L-similarity sets are $\{\{q_1, q_3, q_6, q_2\}, \{q_0, q_5\}, \{q_4\}\}$.

4.4 Construction of the Minimal Cover Automaton

In this section, we give the algorithm constructing the resulting DFCA from the live canonical dissimilar sequence and the L-similarity sets we get in the previous section. The idea is to merge those states in the same L-similarity sets in the prefix tree acceptor. The algorithm is formally presented as follows.

Algorithm 4.3 DFCA(PT(L), S_p, d(p)), ($p \in Q$)

i=0

T=Q

While $T \neq \phi$ {

get $p \in T$ with latest visit time in the previous algorithm level(p),

let $q_i = p$

T=T-S_{qi}

i++

}

m=i

$Q'=\{q_0,\ldots,q_{m-1}\}$, $F'=\{q_i \mid S_{qi} \cap F \neq \phi\}$, $\delta'(q_i, a) = \delta'(q_j, a)$ iff there exists $p \in S_{qi}$ $r \in S_{qj}$, $\delta(p, a) = \delta(r, a)$.

if $q_i \in Q$, $\delta(q_i, a) \notin Q$, $\delta'(q_i, a) = q_{-1}$. for all $a \in \Sigma$, $\delta'(q_{-1}, a) = q_{-1}$, $Q'=Q' \cup \{q_{-1}\}$,

Output $A=(\Sigma, Q', q_0, \delta', F')$

Example 4.5 The resulting minimal DFCA from Example 4.1 is shown in Figure 4.7.

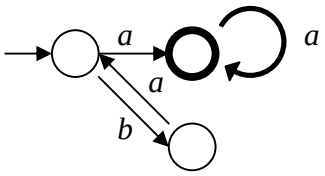


Figure 4.7 Resulting minimal DFCA

4.5 Correctness and Complexity

To summarize our algorithm to get the minimal DFCA from L, we construct the prefix tree acceptor PT(L) from the given strings of a finite language L. Then, we use recursive rules to find live canonical dissimilarity sequence. Finally, we merge those states in the same L-similarity set.

Lemma 4.8 Let $A=(\Sigma, Q, q_0, \delta, F)$ be a DFCA of a finite language L. Let $s, p, q \in Q$ such that level(s)=i; level(p)=j, level(q)=m; $i \le j \le m$. The following statement is true: If $s \sim p$, $s \sim q$, then $p \sim q$. [Campeanu 2001]

Lemma 4.9 In our algorithm, (1) the output $A=(\Sigma, Q', q_0, \delta', F')$ is a DFCA of L, and (2)$[q_{-1}, q_0,\ldots,q_{m-1}]$ is a canonical L-dissimilar sequence. ($[q_0,\ldots,q_{m-1}]$ is a canonical L-dissimilar sequence, if q_{-1} does not exist.)

PROOF The algorithm first constructs PT(L), which exactly accepts L. By Lemma 4.4, 4.5, 4.6 and 4.7, for each $p \in Q$, the algorithm finds a partition $\pi = \{S_1,\ldots, S_n\}$ of L such that S_i is an L-similarity set. Lemma 4.1, 4.2 and Theorem 4.3 set $S^- = \{p | p \in \Sigma^*, p \notin Q\}$, and S^- is an L-similarity set. By definition of L-similarity set, for any $p,q \in S$, $p \sim q$, *i.e.*, $p \sim q$ if for all $w \in \Sigma^*$, $|pw| \le l$ and $|qw| \le l$, $pw \in L$ iff $qw \in L$. By Lemma 4.8, the algorithm only merges such partitions, $L(A) \cap \Sigma^{\le l} = L(\text{PT}(L)) = L$. Then the first

part holds. Since $q_i \in S_{qi}$, $q_i \not\sim q_j$, $i \neq j$, and $\pi = \{S_1, \ldots, S_n, S^-\}$ ($\pi = \{S_1, \ldots, S_n\}$, if S^- does not exist) is a partition of Σ^*. Hence, the second part holds.

Q.E.D.

Corollary 4.2 The output A of our algorithm is a minimal DFCA of L.

By Corollary 4.2, the correctness of our algorithm is proved.

Theorem 4.4 In our algorithm,

(1) the construction of the prefix tree acceptor has complexity $O(n)$,

(2) finding live canonical dissimilarity sequence has complexity $O(n^2)$,

(3) constructing minimal DFCA from the live canonical dissimilarity sequence and corresponding L-similarity set has complexity $O(n)$,

where n is the input length.

PROOF In (1), we only need to scan the input once, so its complexity is $O(n)$. The number of states in PT(L) is $O(n)$. (3) needs to scan each state at $O(1)$, so (3) has complexity $O(n)$. In (2), for each D_i, since every state is calculated with all states in list_i at $O(1)$ and will be added in the list_i after calculated, therefore, for all $p \in D_i$, the running time is $0+1+\ldots+(\#D_i-1)=O((\#D_i)^2)$. So (2) has running time of $O(\Sigma(\#D_i)^2)$. In the worst case, if there only exists one D_0 , $\#D_0=n$, (2) has complexity $O(n^2)$. Exactly, (2) needs $0+1+\ldots+(n-1)$ operations, which sum up to $(n-1)n/2$.

Q.E.D.

4.6 Experimental Results

The algorithm for learning constructing minimal deterministic cover automata is implemented and tested through experiments. The data set is the Splice-junction Gene Sequences Database, which includes gene sequences from classes "ei" (25%), "ie" (25%) and Neither (50%) on http://www.ics.uci.edu/ ~mlearn/MLSummary.html. One experiment is designed to test the advantage of using DFCA instead of traditional DFA. Strings are given as input for the algorithm to construct a minimal DFCA. For comparison, a DFA is constructed by converting a prefix-tree automaton into its equivalent minimal DFA. Figure 4.8 shows the percentages of states in the resultant automata saved by using DFCA instead of DFA.

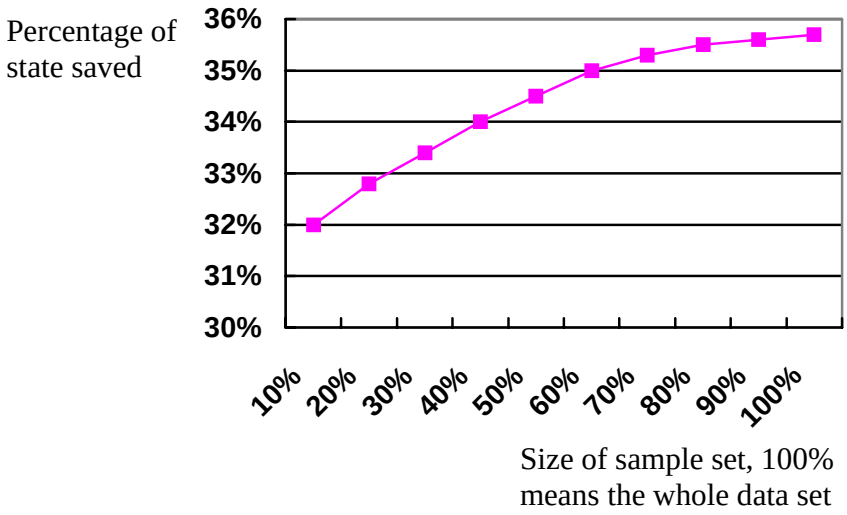


Figure 4.8 Percentages of states saved by using DFCA

It can be seen that more than 32% states are saved by using DFCA. The figure also shows that the size reduction increases when the sample size increases. This result shows that the DFCA is more efficient for larger sample sets. The figure also shows that the changing rate of the size reduction decreases when the sample size increases.

Figure 4.9 shows the running time of our algorithm. It can be seen that this example does demonstrate polynomial running time as predicted by Theorem 4.4.

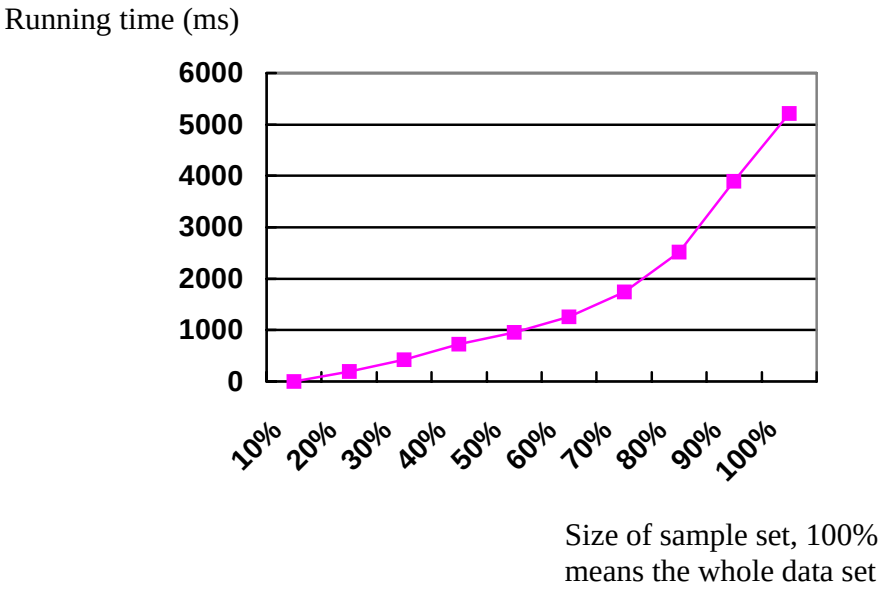


Figure 4.9 Running time of learning minimal DFCA algorithm.

4.7 Conclusions

To reduce the learned model size for single class learning, cover automata is used as learning model. We proposed an algorithm to construct minimal DFCA from strings of a finite language. The algorithm is efficient and has complexity $O(n^2)$, where n is the length of the input. The correctness is proved by theorems. Experiments show that using DFCA for regular language learning, the algorithm reduced the size of the resulting automata by over 32%. We have addressed reducing learned model size for structure only models so far. Next, we are going to extend our approach to handle stochastic models.

.

CHAPTER

5

Inference using Recurrent Neural Networks

To construct efficient models for mixed *k*-th order Markov chains, we present an approach based on *recurrent neural network* (RNN) construction, using statistical property captured through "purity value" to improve efficiency. Here, a *Recurrent Neural Network* (RNN)[Omlin 1996] is a neural network with a "context" layer added to the structure. The inputs are given at each time step and the previous contents of the hidden layer are saved by the context layer, which are then used for feedback into the hidden layer in the next time step. A *Markov Chain* [Rabiner 1989] is a sequence of random variables X_1, X_2, X_3, ... with the Markov property, which means that given the present state, the future and past states are independent, *i.e.*,

$$\Pr(X_{n+1} = x_{n+1} | X_n = x_n, \ldots, X_1 = x_1) = \Pr(X_{n+1} = x_{n+1} | X_n = x_n).$$

A *k*-th *ordered Markov chain* is where

$$\Pr(X_n = x_n | X_n = x_n, \ldots, X_1 = x_1) = \Pr(X_{n+1} = x_n | X_{n-1} = x_{n-1}, X_{n-2} = x_{n-2}, \ldots, X_{n-k} = x_{n-k}).$$

In k^{th} -order Markov Chain, each state transition depends on previous *k* states. We give the nodes the power of counting, and hence call such nodes *processing nodes* in our model. As common with many RNN construction methods, output nodes are activated

by the feedback of some hidden nodes. In addition, at each time step, hidden nodes are copied to *context nodes*, which are used further in the networks. Each context node is constructed based on the context nodes representing less context length. We make this approach more efficient by using "purity score" based pruning. We apply our method to do classification on the Splice-junction Gene Sequences Database. We use 10 fold cross validation to test our method. Based on the experimental results, we further discuss the balance between accuracy and storage cost.

5.1 Construction of the Model

. We now introduce our model through the construction of a recurrent network. We first use our network to learn short sequence patterns. When a shorter pattern is considered to be 'pure' enough, the longer sequence containing the short pattern (as a sub-pattern) is no longer considered, because the short pattern suffices for classification.

An example is shown in Figure 5.1. The first layer is the input layer, and contains the input node I^t. The input I^t represents one symbol in the alphabet Σ of a given sequence. At time t, one symbol in the sequence is given as an input to the recurrent network, and at the next time t+1, the next symbol in the sequence is given as input. Suppose Σ={a, b ,c}, in our example, three nodes a^t, b^t and c^t are connected with the input node I^t. Then, the corresponding node among a^t, b^t and c^t is activated. For example, if the input is a, then the node a^t is activated.

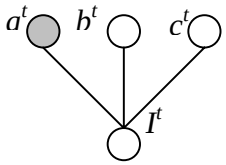


Figure 5.1 First step of recurrent network construction

The construction of the recurrent network contains the following steps:

(1) Let each node be denoted by $s_{i,j}^t$, where t is the time that the node represents, i is the length of the pattern the node represents and j is the number according to the alphabet order among the nodes with the same i and t. For example, in Figure 5.1, the node of a^t is denoted $s_{1,1}{}^t$, b^t is denoted $s_{1,2}{}^t$ and b^t is denoted $s_{1,3}{}^t$. Each node in our model has two counting parameters, one is *positive count* and the other is *negative count*. Each count records the number of appearances in positive sequences or negative sequences, respectively. We call a sequence as *positive sequence* if it belongs to the class which we are interested in, and a sequence as *negative sequence* otherwise. The positive count and negative count are calculated in this way: we take an symbol from the sequence, and activate the node representing the same symbol, *i.e.*, set $s_{1,j}^t = 1$. For example, if the input is a, then a^t is activated. If a node is activated during the training by a positive sequence, then the positive count increases one; if the node is activated during the training by a negative sequence, then the negative count increases one. For a node $s_{i,j}^t$, we denote its positive count $n_{i,j}^{t+}$ and negative count $n_{i,j}^{t-}$. In the example shown in Figure 5.1, after the first scan, the network will capture the frequencies of a, b, and

c in both positive and negative training sequences.

After we have all the counts, we calculate the scores of the nodes using (5.1). The scores will be used for classification after the construction of the network.

$$S_{i,j}{}^{t} = lg(Pr(s_{i,j}{}^{t}|P)/Pr(s_{i,j}{}^{t}|N)). \quad (5.1)$$

where, P is the set of positive samples, and N is the set of negative samples, $Pr(s_{i,j}{}^{t}|P)$ denotes the probability of activating node $s_{i,j}{}^{t}$ in the training of a positive training sequence, while $Pr(s_{i,j}{}^{t}|N)$ denotes the probability of activating node $s_{i,j}{}^{t}$ in the training of a negative training sequence.

In practice, we use the weighted frequencies to represent the probabilities.

$$Pr(s_{i,j}{}^{t}|P) = n_{i,j}{}^{t+} / n_P. \quad (5.2)$$

$$Pr(s_{i,j}{}^{t}|N) = n_{i,j}{}^{t-} / n_N. \quad (5.3)$$

where, n_P and n_N denote the number of positive and negative sequences respectively. Thus, we do not directly make use of some well-known statistical parameters. However, using the previous "history" of activation of the nodes, we define the weighted frequencies in (5.2), (5.3). The *lg* makes the score addable, and simplifies the result from normalization factors.

(2) Before further constructing the nodes for longer patterns, we do pruning. The pruning is done based on a purity threshold *tr*. This *tr* can be determined by the memory space, which we will give the detail in Section 5.4. The purity of a node is $max(Pr(s_{i,j}{}^t|P)/(Pr(s_{i,j}{}^t|P) + Pr(s_{i,j}{}^t|N), Pr(s_{i,j}{}^t|N)/(Pr(s_{i,j}{}^t|P) + Pr(s_{i,j}{}^t|N))$. If a node's purity is higher than *tr*, then we take the node as a 'pure' node and do not use in further extension, because it is much more likely to appear in positive sequences (or negative sequences) than the opposite class. For example, in the same example shown in Figure 5.1, we have $a^t(n^+: 3, n^-: 1)$, $b^t(n^+: 1, n^-: 2)$, $c^t(n^+:5, n:4)$ and $tr = 75\%$, then, the node a^t is considered pure and b^t, c^t are not pure.

(3) After the pruning, nodes representing longer patterns are constructed. For each new found non-pure node, a context node representing time *t*-1 on the same pattern is created. Then, for each combination of the newly created context nodes and non-pure nodes representing the symbols in alphabet, a new node is created. We continue the example in Figure 5.1. We have pruned a^t in the previous step. Then, b^t and c^t are the newfound non-pure nodes, therefore, context nodes b^t and c^t are created as shown in Figure 5.2.

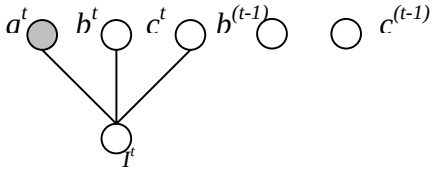


Figure 5.2 Context nodes creation in the recurrent network

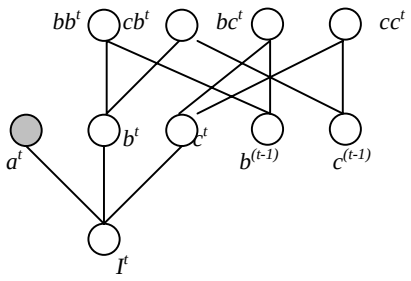


Figure 5.3 Construction of nodes for longer patterns

Then, nodes representing longer patterns are created, with their inputs from corresponding context nodes. As shown in Figure 5.3, in the same example, bb^t is created by joining b^t and b^{t-1}, cb^t is created by joining b^t and c^{t-1}, etc.

The function for activating the newly created nodes is determined at this step. For cb^t in our example, its function is:

$cb^t = \Phi(b^t + c^{t-1} - 2)$, where $\Phi(x) = 1$, when $x \geq 0$, $\Phi(x) = 0$ otherwise. Generally, the function of node ${s_{i,j}}^t$ is:

${s_{i,j}}^t = \Phi({s_{1,j}}^t + {s_{i-1,m}}^t - 2)$, where the connection of the patterns represented by ${s_{i-1,m}}^t$ and ${s_{1,j}}^t$ is ${s_{i,j}}^t$.

(4) After the construction of the new nodes, the sequences are input to the network again. This time, the network uses the inputs in this way: at time t, I^t gets the input

symbol from the sequence, activate nodes according to the functions, then all nodes representing patterns in time *t* are feedback to the context nodes representing the same patterns but at time *t*-1. The feedback function is: $s_{i,j}{}^{t-1} = s_{i,j}{}^{t}$. Then, the next symbol is input. If a 'pure' node is activated, the network resets all activated nodes to inactivated ones, since the shorter patterns should have higher priority than longer patterns. If not, the context nodes are feedback. Still, if a 'pure' node is activated, the network is reset. If the network activates a newly created node, the counts are accumulated. Suppose we have a positive sequence of '*cba*', which is input to the network in Figure 5.3. The first symbol is *c*, so c^t is activated. Then, by feedback, c^{t-1} is activated. The next symbol is *b*, b^t is activated, since we have both c^{t-1} and b^t activated, the node cb^t is activated. Since cb^t is a newly created symbol, the positive count of the node is increased by 1. This time, the feedback will activate b^{t-1}. The next symbol is *a*, which will activate a^t, a pure node, and then the network is reset. After these, again, the scores and purities are calculated; 'pure' nodes are pruned. After the 2nd scan, the recurrent neural network is like the one shown in Figure 5.4. The shaded nodes are pure nodes.

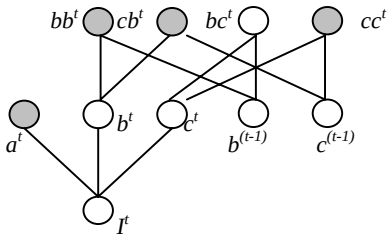


Figure 5.4 Recurrent network after the second scan

After this, we repeat steps (1)-(4), until all the new constructed nodes are 'pure', or

after the k^{th} scan, where the number k is the length of the longest sequence pattern we want to capture.

In our example, the context node bc^{t-1} is created. Then, new nodes of longer patterns are created as bcb^t and bcc^t. In the third scan, the counts of the newly created two nodes are counted, , purity and score are calculated and then pure nodes are marked. This time, we do not have non-pure node, therefore the construction ends, and the resultant network is shown in Figure 5.5.

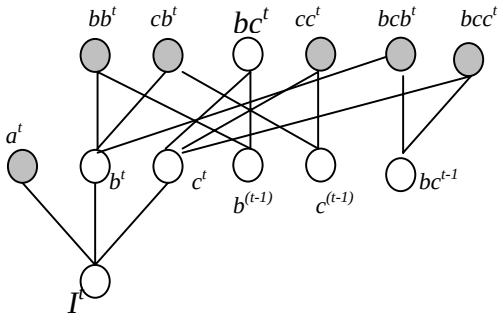


Figure 5.5 Resultant recurrent network

The steps in the above construction are stated in the following Algorithm 5.1.

Algorithm 5.1 Constructing RNN for Markov Chains

Input: sample string set, length limit *k*, and purity level limit *p*

Create an input node for the empty recurrent neural network *N*.

For each symbol of the sample alphabet, create a node in the hidden layer, connect the nodes with the input node.

For *i*=1 to *k* {

Scan sample *S*, calculate the count that a newly created hidden layer node is activated by each string class.

Mark the nodes with purity level higher than *p* as 'pure node'

Move the newly created hidden layer nodes as context nodes

If all nodes are marked as 'pure node',

Then Exit the For-Loop

Else{

Add new nodes with connections to hidden layer through connecting the strings represented by the context layer and non-pure nodes representing alphabet.

}

}

Output *N*

The network has $O(|\Sigma|^k)$ nodes. The time complexity is $O(k\ n\ |\Sigma|^k)$ where n is the input length. Here k and $|\Sigma|^k$ are constant if the model has been determined, so the time complexity is only linear with the input length n.

5.2 Recurrent network for classification

After the construction of the resultant network, it can be used for classification of sequences. The running is similar to step 4 in the construction. The difference is that, when the network activates a pure node, it will accumulate the score for the sequence;

and when the network activates a node representing k length pattern, instead of accumulating the counts, the network will accumulate the score and reset. The classification is explained in the following steps:

Algorithm 5.2 Classification by RNN for mixed Markov Chains

Input: unclassified sequence

For each symbol in the unclassified sequence {

Activate a node with a matching symbol

If all nodes that connected to a hidden layer are activated
Then, activate the hidden layer node

If the activated hidden node is a 'pure' node,
or a node representing length k pattern

Then{ calculate the score S for sequence as $S = S + S_{i,j}$
Reset all activated nodes to inactive. }

Else Copy the activating result of the hidden layer to the context layer.

}

Output positive, if S is positive, and negative otherwise.

We observe in the construction that, each node represents a sequence pattern in the network; the counts exactly record the appearances of each pattern; the score gives the comparison between the probabilities of an observed pattern in positive and negative sequences. Therefore, like in the Markov Chains model, our model generates probability based scoring over sequences, and can be used for sequence classification.

5.3 Distribution patterns

In this section, we discuss the concept of distribution patterns. In previous sections, we presented our model and methods to get the sequence patterns of different lengths from training sequences. When we put a set of sequences as the training set, the trained recurrent network is able to capture all sequence pattern information and their frequency. For each sequence pattern, we have a probability parameter. Hence, the recurrent network has the probability parameters for all typical sequence patterns. Therefore the recurrent network can be regarded as a pattern distribution summary of the training sequence set.

With this concept, we extend our method. Instead of inputting all negative and positive sequence set samples to train the recurrent network, a recurrent network is trained for every given sample set. Since the score is not available this time, we use the number of activations instead. Then, each sequence set has a corresponding pattern distribution network.

With these neural networks for each class, we do comparison between the neural networks instead of going into details of the sequences. This results in a large savings in the I/O cost. We now give the algorithm for such kind of comparison.

Algorithm 5.3 Markov Chain RNN Comparison

Input: RNN RNN_1 and RNN_2 for Markov Chains, difference level d

For each node in RNN_1 and RNN_2 {

Comparing the score of each node in RNN_1 and RNN_2 representing the same sequence pattern

If one score over the other one is greater than difference level d

Highlight this node as a difference mark

}

A new neural network is constructed by attaching the counts of each node in RNN_1 and RNN_2.

The nodes in the new neural network RNN_3 corresponding to the highlighted nodes are marked as 'pure' nodes, while the other nodes are non-pure.

Output RNN_3

5.4 Pruning methods

For the method introduced in Section 5.3, we use purity based pruning methods. The purity threshold *tr* affects the pruning rate and further affects the accuracy, since it causes loss of information for some nodes with purity around threshold. This purity threshold can be determined by the available memory space and the maximum length of sequence pattern.

We first start with a very low pruning rate and very low purity threshold. If the

memory cannot hold the network, we increase the purity threshold. And if the memory can hold, we then decrease the purity threshold. In this way, the best purity threshold will be found. Our model can be used to find itemsets patterns in transaction data. In this situation, the pruning factor is the frequencies or support rate of each pattern. Therefore, we are able to use frequency based pruning method.

We first order the symbols by their frequencies. Then, some symbols under a certain frequency threshold are pruned. The remaining symbols are then sorted based on their frequencies. While constructing neural network, in the copying activation step, the context nodes are extended according to the order.

For example, suppose the items are {*a*, *b*, *c*}, by their frequencies, their order from the most frequent item to the most infrequent one is (*a*, *b*, *c*). This example is illustrated in Figure 5.6.

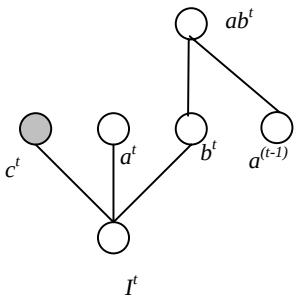


Figure 5.6 Recurrent network in itemset-oriented application.

In the first order nodes, the node of c^t is an example of pruned node because of not enough frequency. Therefore, after the first phase, only *a* and *b* are extended to the

next layer. Because this is an item set-oriented approach, while copying, the most infrequent node, in this case *b*, is not copied. In this example, we only have two nodes left. Generally, we will copy all frequent items other than the most infrequent node. Then, the second order recurrent neural network is constructed. New combination nodes are created based on the frequency order. In this example, while copying the context nodes, ab^t is constructed, but ba^t is not constructed. Note that, the node is not used for constructing context node, but still used in the neural network to capture first order information.

Figure 5.7 shows a larger example, in which we illustrate the construction of the second order nodes.

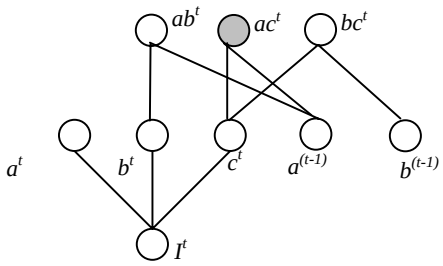


Figure 5.7 Two-items nodes construction in recurrent neural network for itemset-oriented applications.

When the application is about itemsets, the previous pruning method is very efficient.

5.5 Experimental Results

We implemented our method of constructing recurrent neural networks from sequences. The method is applied to do classification on the Splice-junction Gene Sequences Database, which includes gene sequences from classes “ei” (25%), “ie” (25%) and Neither (50%) on http://www.ics.uci.edu/ ~mlearn/MLSummary.html. Each time, for the three classes, we use the sequences in that class as positive sequences and those sequences from the other two classes as negative sequences.

We use 10 fold cross validation to test our method. The following figure shows the accuracy under different *k*-levels. This case is about sequence classification, therefore, the pruning method is purity based.

Figure 5.8 shows the logarithm (with base 10) of the number of nodes of the constructed recurrent neural network different purity thresholds. It can be seen that as purity threshold decreases, the size of recurrent neural network constructed is greatly reduced. However, this is at the cost of accuracy. From Figure 5.9 we can see that when the threshold drops, the accuracy drops from 80% to 70%. This is because that, more detailed information will get more accuracy of describing the sequences. Higher purity limitation means less pruning and therefore means more cost in time and space. Hence, there is a balance between the accuracy and efficiency. In practice, we can use a purity threshold where the reduction of the model size is slowing down or an expected model size is set according to performance requirement.

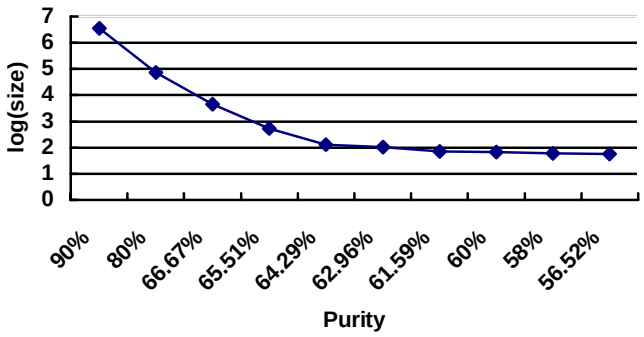


Figure 5.8 Model size under different purity thresholds

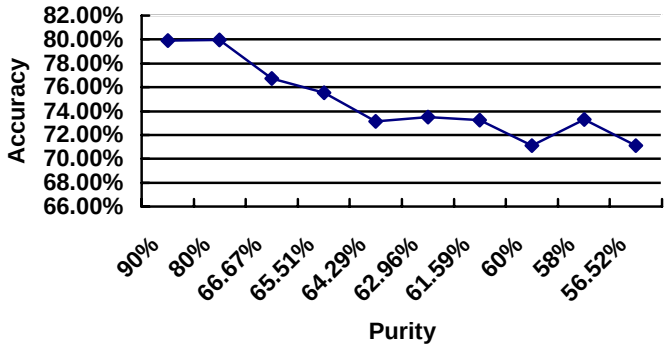


Figure 5.9 Accuracy under different purity thresholds

5.6 Conclusions

We have presented an explicit construction method to train recurrent networks for mixed length sequence patterns. Our method uses stochastic information, and therefore

is noise tolerant. However, unlike existing stochastic inference, the structure of our model is dynamically constructed during the training process instead of an arbitrary one. Our method also captures sequence patterns with different length in the training set while ignoring non-typical relationships. It uses pruning to avoid falling into trivial separations and prevent higher complexity. The networks, at least to limited extent, can capture pattern distribution. We discussed the way of using pattern comparison of a summarized pattern distribution recurrent network to construct classification-oriented networks. Pruning methods are discussed. We suggested a pruning method for itemsets based applications.

In the experimental results, we show that pruning by lower purity threshold can reduce the learned model size greatly. For the tradeoff between accuracy and model size, one can choose a purity value where further reduction of the value helps little to model size or where the model size is already acceptable according to performance requirement. In the next Chapter, we further extend model reduction method on context-free languages learning.

CHAPTER

6

Inference using Profile Based Alignment Learning

We use Profile Based Alignment Learning framework to address the two problems of ABL described in Chapter 2. An overview of our framework is given in Figure 6.1. The input data are sample sentences of the target language, with or without structural information. A simple context free grammar is built by creating a production rule with left hand side as the grammar start symbol and right hand side as each sentence. Therefore, the built grammar can generate the given sample set exactly. Starting from this simple context-free grammar, pairwise alignment is done for each pair of the right hand side of the productions. According to the alignments, constituent pairs aligned in the same slots are given *slot alignment score*s, which represent how often the constituents are in the same slot. *Alignment profile similarities* are further calculated based on the slot alignment score, which is used for similarity measures. This similarity measure is then used as the similarity of the constituents for the calculation of further alignment scores. In this way, previous alignments are changed. If there is

no further change to the alignments, then the grammar will be updated according to the alignments. Otherwise the similarities will be calculated again. The above steps are repeated until there is no further change to the grammar, *i.e.*, no more change to the grammar production rules. Finally, the learned grammar is given as the output.

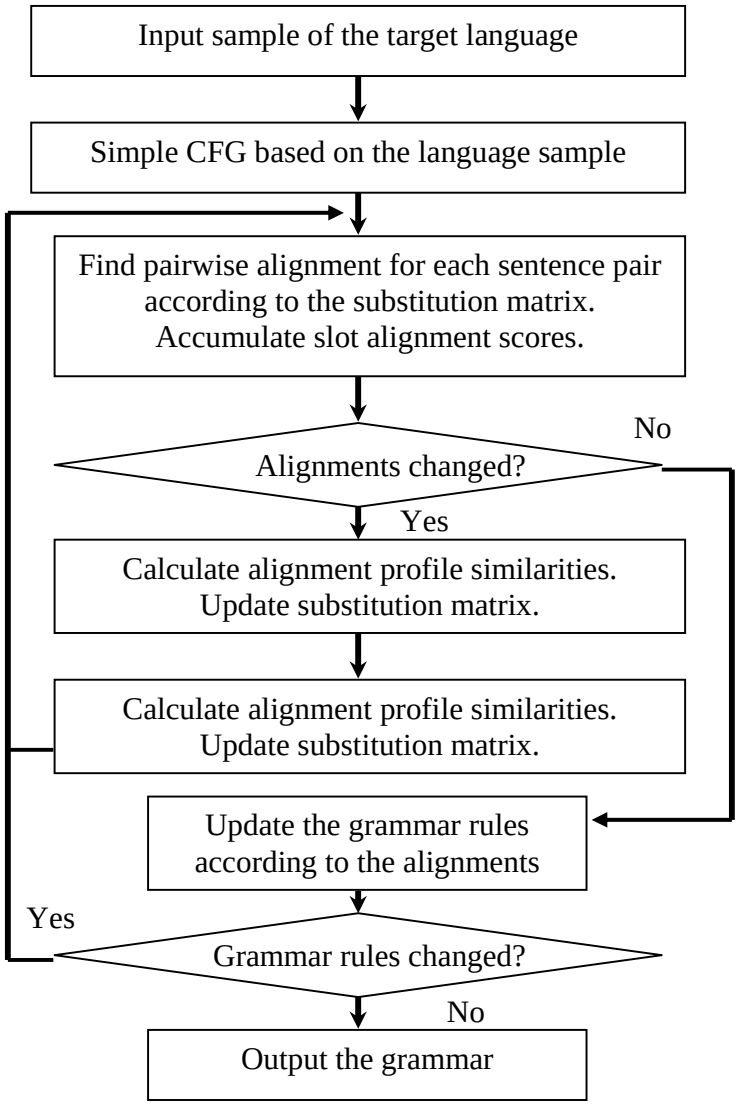


Figure 6.1 Overview of the PBAL framework

6.1 Notations

We define an alphabet as a finite set of symbols. A string over an alphabet Σ is a finite ordered sequence of symbols from Σ. The length of a string α is the number of symbols in the string, with repetitions counted, and it is denoted by $|\alpha|$. (e.g., $|aabbc| = 5$). The empty string, denoted by λ, is the string of length zero. If x and y are strings, then xy is the connection of the strings. If A and B are sets of strings, then $AB = \{xy \mid x \in A \text{ and } y \in B\}$. Giving an string set A, we define $A^0 = \{\lambda\}$, $A^{n+1} = AA^n$, $A^* = \{x \mid x \in A^n, n = 0,1,2\ldots\}$.

Define language $L \subseteq \Sigma^*$. A context-free grammar (CFG) G is a 4-tuple (N, Σ, P, S), where N is the set of nonterminals, Σ is the alphabet, P is the set of productions of the form $A \rightarrow \alpha$, and $S \in N$ is the start symbol. The symbols in the alphabet are also called terminals. We use capital letters $A, B, \ldots$ to denote nonterminals, lowercase letters $a, b, \ldots$ in the beginning of the alphabet to denote terminals, and Greek letters $\alpha, \beta, \ldots$ to represent strings in $(N \cup \Sigma)^*$.

Let $L(G)$ denote the language generated by grammar G. A language L is a context-free language (CFL) if there exists a context-free grammar G such that $L=L(G)$. A CFG G is stochastic CFG (SCFG) if there is an assignment of weights, each weight being between zero and one inclusive, to the right-hand sides of productions such that for every nonterminal A, the weights of the right-hand sides of all the productions with A on their left-hand sides sum to 1. A production p with A on the left-hand side α on the

right-hand side with probability $pr(p)$ is denoted by $A \rightarrow \alpha$ ($pr(p)$). For a production $p \in P$, RHS(p) denotes the right hand side of p, LHS(p) denotes the left hand side of p. A fuzzy set A' on X is given by $A' = \{<x, \mu_{A'}(x)> | \ x \in X\}$, where $\mu_{A'}(x) \in X \rightarrow [0, 1]$ is the membership function of $x \in X$ in A'.

6.2 The Profile Based Alignment Learning System

6.2.1 Input and pre-processing

The input data to our PBAL are the strings from the target language. If the application needs to use as alphabet symbol as the processing unit, then the symbols are used as terminal symbols in CFG. In other applications, other language units like words in natural languages are used as the terminal symbols. Accordingly, for each sample sequence, represented as a string α of terminal symbols, a production is added to the production set as $S \rightarrow \alpha$, where S is the starting symbol. For example, suppose the following texts are in the given sample set.

Example 6.1 Two sample paragraphs for illustrating initial step in PBAL

Sample text 1:

All IOC members had rehearsed the voting procedure in Singapore before the first round and had about a minute to choose. The members had to press keys representing the cities and then confirm or cancel their choice by pressing a separate key.

Sample text 2:

London came out on top when the announcement was made by IOC President Jacques Rogge in Singapore. This was the fourth bid from Britain. London will become the first city to have hosted the Olympics three times.

Based on the above sample texts, the following production rules are created.

$S \rightarrow$ [*All*] [*IOC*] [*members*] [*had*] [*rehearsed*] [*the*] [*voting*] [*procedure*] [*in*] [*Singapore*] [*before*] [*the*] [*first*] [*round*] [*and*] [*had*] [*about*] [*a*] [*minute*] [*to*] [*choose*] [*The*] [*members*] [*had*] [*to*] [*press*] [*keys*] [*representing*] [*the*] [*cities*] [*and*] [*then*] [*confirm*] [*or*] [*cancel*] [*their*] [*choice*] [*by*] [*pressing*] [*a*] [*separate*] [*key*]

$S \rightarrow$ [*London*] [*came*] [*out*] [*on*] [*top*] [*when*] [*the*] [*announcement*] [*was*] [*made*] [*by*] [*IOC*] [*President*] [*Jacques*] [*Rogge*] [*in*] [*Singapore*] [*This*] [*was*] [*the*] [*fourth*] [*bid*] [*from*] [*Britain*] [*London*] [*will*] [*become*] [*the*] [*first*] [*city*] [*to*] [*have*] [*hosted*] [*the*] [*Olympics*] [*three*] [*times*]

If the structural information is available, then a simple context-free grammar can be constructed directly from the information. The constructed grammar has the same hierarchical structure with the given sample. In the following example text, periods

and paragraphs are used as structural information.

Example 6.2 Illustration of the use of basic structure like paragraphs and sentences for the formation of grammar rules

London came out on top when the announcement was made by IOC President Jacques Rogge in Singapore.

This was the fourth bid from Britain. London will become the first city to have hosted the Olympics three times.

Making use of the basic structure of paragraphs and sentences, the production rules of a simple grammar are given below. The nonterminals Sn_1, Sn_2, Sn_3 represent sentences while the nonterminals P_1 and P_2 represent paragraphs.

$Sn_1 \rightarrow$ [*London*] [*came*] [*out*] [*on*] [*top*] [*when*] [*the*] [*announcement*] [*was*] [*made*] [*by*] [*IOC*] [*President*] [*Jacques*] [*Rogge*] [*in*] [*Singapore*]

$P_1 \rightarrow Sn_1$

$Sn_2 \rightarrow$[*This*] [*was*] [*the*] [*fourth*] [*bid*] [*from*] [*Britain*]

$Sn_3 \rightarrow$ [*London*] [*will*] [*become*] [*the*] [*first*] [*city*] [*to*] [*have*] [*hosted*] [*the*] [*Olympics*] [*three*] [*times*]

$P_2 \rightarrow Sn_2\ Sn_3$

$S \rightarrow P_1 P_2$

To calculate the stochastic information in the grammar, a count is attached for each production rule. The count is the number of times the corresponding production rule used in the sample. In the preprocessing phase, a scan is done on all sample texts. Duplicated sample texts (sample texts that have exactly the same copy in the whole sample set) are deleted. The number of the copies of the duplicated sample text including the original one is counted as the *duplicate number* of the corresponding sample text. When the CFG is constructed, the count of all the production rules created is set to the duplicate number of the corresponding sample text. The count for a production rule p is denoted by $C(p)$. We illustrate this construction in the following example.

Example 6.3 Use of "count" $C(p)$ for a production rule p.

Sample text 1:

London came out on top when the announcement was made by IOC President Jacques Rogge in Singapore.

Sample text 2:

This was the fourth bid from Britain. London will become the first city to have hosted the Olympics three times.

Sample text 3:

This was the fourth bid from Britain. London will become the first city to have hosted the Olympics three times.

Then the production rules and their counts are as follows.

p_1: $Sn_1 \rightarrow$ [*London*] [*came*] [*out*] [*on*] [*top*] [*when*] [*the*] [*announcement*] [*was*] [*made*] [*by*] [*IOC*] [*President*] [*Jacques*] [*Rogge*] [*in*] [*Singapore*], ($C(p_1) = 1$)

p_2: $S \rightarrow Sn_1$, ($C(p_2) = 1$)

p_3: $Sn_2 \rightarrow$[*This*] [*was*] [*the*] [*fourth*] [*bid*] [*from*] [*Britain*], ($C(p_3) = 2$)

p_4: $Sn_3 \rightarrow$ [*London*] [*will*] [*become*] [*the*] [*first*] [*city*] [*to*] [*have*] [*hosted*] [*the*] [*Olympics*] [*three*] [*times*], ($C(p_4) = 2$)

p_5: $S \rightarrow Sn_1\ Sn_2$, ($C(p_5) = 2$)

Let P be the production set. The probabilities of the production rules, denoted by $pr(p)$ for production p, can be calculated from the counts:

$$pr(p) = \frac{C(p)}{\sum_{q \in P, LHS(p)=LHS(q)} C(q)} \qquad (6.1)$$

Suppose the given sample set contains only sample texts 1,2,3 and no other texts, then $pr(p_5)$ is given by $C(p_5) / (C(p_5) + C(p_2)) = 2/(2+1) = 2/3$.

After the preprocessing, the input to PBAL is the context-free grammar, including its production rules with counts, alphabet, nonterminals and start symbol.

6.2.2 Pairwise Alignment Learning

First, we briefly introduce a simple version of dynamic programming (DP) for pairwise alignment [Needleman 1970]. As shown in Example 6.4, for symbol *A* and *B* in the two strings "*TGTAT*" and "*TGTBT*", if *A* and *B* are aligned as a match, then an alignment score is given to the alignment, which is a preset value representing the similarity of the symbols *A* and *B*, denoted by $s(A, B)$. Such a value is called a substitution score. A matrix containing such substitution scores between all symbols is called a *substitution matrix*.

Example 6.4 An example to illustrate "alignment score" $s(A, B)$

T	*G*	*T*	*A*	*T*
T	*G*	*T*	*B*	*T*

Alignment score = $s(T, T)+ s(G, G)+ s(T, T)+ s(A, B)+ s(T, T)$

If a symbol *A* is pairing with a gap, then a gap penalty, denoted by *g*, is given to the alignment. Example 6.5 illustrates this gap penalty.

Example 6.5 An example to illustrate the "gap penalty" *g*.

T	*G*	*T*	*A*	*T*
T	*G*	*T*	-	*T*

Alignment score = $s(T, T) + s(G, G) + s(T, T) + g + s(T, T)$

A DP matrix (shown in Figure 6.2) for string γ_1 of length $m+1$, and γ_2 of length n is a matrix with m rows and $n+1$ columns.

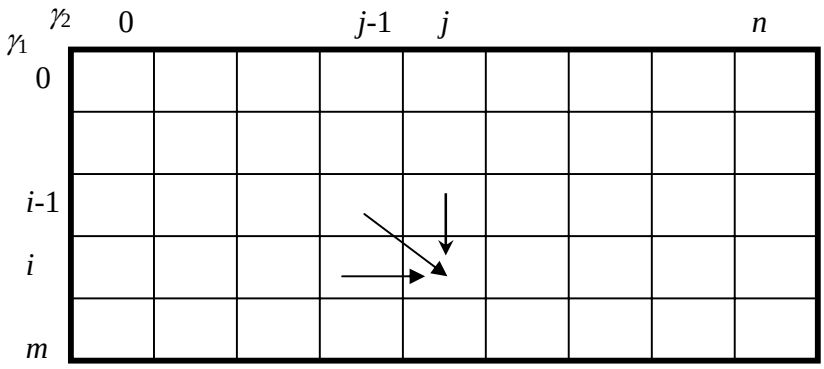


Figure 6.2 DP Matrix

To find the best alignment between two sequences, DP algorithm uses recurrence relations. The initial values are set in the following way. The cell (0, 0) is given an initial score 0. Starting from (0, 0), each step can be the following three possible moves:

(i) moving right,

(ii) moving down,

(iii) moving right down.

Moving right or moving down means insertion or deletion and therefore causes a penalty g to the sum of the alignment score, while moving right down means substitution and therefore adds $s(\gamma_1[i], \gamma_2[j])$ to the sum. The recurrence works in the following way. When selecting the move to (i, j), one can move from $(i-1, j)$ with gap penalty to the score at $(i-1, j)$, or from $(i, j-1)$ with gap penalty to the score at $(i, j-1)$. Or, one can move from $(i-1, j-1)$ with additional score $s(\gamma_1[i], \gamma_2[j])$ to the score at $(i-1, j-1)$. The best move is the one with the best resultant score, and therefore selected. In this way, when the best move to (m, n) is selected, the best alignment score is reached. By tracing back all the best selections made from (0, 0) to (m, n), the best alignment is produced.

After the preprocessing as described in Section 6.2.1, PBAL does the pairwise alignment for production. Each right hand side of production rule is aligned with each other one in the input context-free grammar. The pairwise alignment is done using dynamic programming. For alignment learning, the goal of the pairwise alignment is to find the maximum match alignment of each sentence pair. Therefore, the alignment scores are the number of matches of the sentence pair. There is no penalty. In the first iteration, elements in the diagonal of the substitution matrix are one, other elements are set to zero, *i.e.*, initially only exact matches are considered as matches. In the substitution matrix, the highest similarity score is 1, and the lowest similarity score is 0.

In alignment learning approaches, inference of grammatical rules starts from the alignments. The information given by these alignments determines the effect of the learning process. We propose additional techniques on the alignment learning to improve the alignment results.

(1) Sentence Similarity

In real application, sentence pairs like the following example generate misleading alignments and thus result in identification of incorrect patterns.

Example 6.6 An example to illustrate "misleading alignments".

[*Thank*] *you for showing me the games* .

[*How are*] *you* .

In Example 6.6, one single common word '*you*' in the two quite different sentences gives misleading information. To solve this problem, we use the "sentence similarity", which represents the similarity measure. We take the "sentence similarity" as the alignment score of the two sentences. Thus, in our framework, the sentence similarity is the sum score of matching pairs of terminal symbols in the two sentences. A threshold, namely sentence similarity threshold (SST) is set. Only those sentence pairs with similarities larger than the SST will be processed. For example, if the SST is 1, suppose there are no other matches, then the previous example (Example 6.6) is not

processed further. When the SST is set to 0, the algorithm does not use SST; hence any sentence pair with positive alignment score is processed. This technique is used for preventing the sentences having common parts by chance to generate misleading information. We compared the original Alignment Based Learning (ABL) of Zaanen [Zaanen 2002a], denoted ABL, and ABL with different SSTs in experiments. Details have been included in Section 6.4.

(2) Relative Sentence Similarity

The similarity measure in the above settings is the alignment score. In that way, longer sentences tend to have higher score. An alternative is to use relative score as measure for sentence similarity. The score is normalized by the lengths of the two sentences. Denote this score *R_Score*. $R\_Score = \frac{2A\_Score}{(len_i + len_j)}$, where *A_Score* is the alignment score, len_i and len_j are the lengths of the two sentences respectively. *A_Score* based method and *R_Score* based method are tested through experiments (Section 6.4). The *R_Score* based methods are shown to be better than the *A_Score* based method.

(3) Dynamic Sentence Similarity

When the SST is set too high, say exceeding the max length of sentences, no alignment is recognized. Even if several patterns are recognized, in the worst cases, the recognized patterns can be incorrect due to the fact that they are not capable of being captured by CFG based rules. Example 6.7 illustrates this case.

Example 6.7 Non-matching sentences with high similarity measure.

they all [have] to stay in order.

this time they are all [going] to stay in order.

In Example 6.7, the sentence similarity is high, but the words "*have*" and "*going*" are definitely not interchangeable. If a CFG rule is created, then the words are made interchangeable, and the pattern is incorrect. This is due to the usage of "*be*" and verbs are beyond CFG, *i.e.*, the form of the verb must be determined by the context and therefore is not context-free. We can create specific simple rules, like never creating a rule for verbs in different forms. This can prevent such mistakes efficiently. However, there is no general method to recognize such non-context-free cases. Therefore, we observe that when the SST is too high, the result is unstable.

Finding a reasonably good threshold value for SST is a solution for this problem. Since the suitable sentence similarity varies in different samples, we designed a way to dynamically change the sentence similarity threshold (SST). The threshold is first set at a high value, for the relative sentence similarity threshold, we first set the threshold to 1 and do alignment, if only several rules or even no rules are found, then the threshold decreases gradually until new rules are generated. In each iteration, the threshold is set high initially. To prevent generating large number of misalignment results, empirically, the lowest threshold allowed is set to 0.5 which ensures at least half of the sentences are structurally in common. We call this technique as *Dynamic SST PBAL*.

(4) Using Similar Constituents as Slot Border

If a word is next to any slot in an alignment, we say that the word is a *slot border*. In the following two sentences,

And what do [*I do with my blue card*]*?*

What do [*you have on your car now*]*?*

the word "*do*" is a slot border. The slot borders are used to identify the grammar rule creation. If a previous alignment gives '*my*' and '*your*' high similarity score in the substitution matrix, then it may find the following more useful structural information.

And what do [*I do with*] *my* [*blue card*]*?*

What do [*you have on*] *your* [*car now*]*?*

Implementation is done to test the effect of using similar words as slot borders. If the similarity score in the substitution matrix of two constituents is no less than a set threshold, denoted by *alignment border similarity threshold* (ABST), then the two constituents are used as slot borders. When ABST is 0, we use any constituent pair with positive similarity as borders; when ABST is 1, only exact matches are used as border, *i.e.*, the similarity scores in the substitution matrix of different words are not used for slot identification and are only used in getting the alignment.

6.2.3 Update Substitution Matrix

Using the alignment results, we update other elements in the substitution matrix; this update is done to improve the alignment further. The main methods used for this update are presented below.

(1) Slot Alignment Score

In alignment based approaches, only exact matches can infer rules. For example, suppose the input set contains the following two sentences:

Jack likes apples.

Jack likes pears.

"*Jack likes*" is then regarded as a common subsentence, while "*apples*" and "*pears*" are regarded as replaceable parts. The system then generates production rules indicating the words '*apples*' and '*pears*' are of similar usage in the structure. The rules can be like:

$S \rightarrow$ *Jack likes A*

$A \rightarrow$ *apples*

$A \rightarrow$ *pears*

However, for subsentences of no common words, no information is used for production rule generation. For example, the two subsentences in the brackets in the following example are considered to be totally different.

Jack likes apples.

Jane loves pears.

In real application language samples, it is quite common for the two sentences to have similar grammatical structure but contain no words in common.

Using lexical or tagging information can be of help, like in the above case, if '*likes*' and '*loves*' are labeled to be a match, then the sentences can be associated as:

Jack [*likes*] *apples.*

Jane [*loves*] *pears.*

The generated rules then can be like:

$S \rightarrow A\ B\ C$

$A \rightarrow Jack$

$A \rightarrow Jane$

$B \rightarrow likes$

$B \rightarrow loves$

$C \rightarrow apples$

$C \rightarrow pears$

However, the problem has not been completely solved. Firstly, words that can be labeled as matching words depend highly on situation, and there are few words that are interchangeable in all situations. Secondly, in some applications, like bioinformatics, detailed tagging information is not available.

To solve this problem, we propose to use the statistical information suggested by the sample itself. Subsentences that are found in the same alignment slot tend to, though not always, have similar grammatical structures. Therefore, if component pairs in the

same alignment slot are marked, then they can help in further alignment. Suppose in the aligning process, we have the following two sentences.

And what do [*I do with my card*]*?*

What do [*you have on your car*]*?*

The constituents "*I do with my card*" and "*you have on your car*" are aligned in the same slot as shown by the brackets. Though there are no word matches in the slot, we can see the similarity between the two structures. From this alignment, we want to record something indicating that

(i) '*I*' is structurally similar to '*you*',

(ii) '*do*' is structurally similar to '*have*' and

(iii) '*with*' is structurally similar to '*on*'.

However, a simple one-to-one match for the words in an alignment slot is not a good solution. Consider the following situation.

And what do [*I do with my blue card*]*?*

What do [*you have on your car now*]*?*

Here, the word-to-word (constituent-to-constituent) alignment is broken. The word '*blue*' and '*car*' is not a desired alignment. If rules like $A \rightarrow$ *blue*, $A \rightarrow$ *car* are generated, then the inferred structure is incorrect.

Moreover, there are situations where even exact matching words are not desired alignment. Like in the following two sentences:

[*Now you have*] *to* [*go backwards*] .

[*I want a ticket from Singapore*] *to* [*Melbourne*] .

the structure indicated by the alignment is incorrect, though the parts containing word '*to*' in the two sentences exactly match each other.

To allow our learning method to take care of such situations, we propose a scoring scheme for the aligned words. The scoring scheme tends to represent the possibilities and frequencies that a pair of words or subsentences should be aligned as a match. For example, suppose we have two sentences in the sample set:

Jack loves apples.

Jimmy likes pears.

Assume that, from some previous calculation, the similarity of the words "*loves*" and "*likes*" are qualified to be considered as a match. Based on this information, the two sentences can be structured like:

[*Jack*] *loves* [*apples*].

[*Jimmy*] *likes* .

(2) Scoring Scheme

We require that the scoring scheme should be capable of representing the alignment frequency and possibility. The idea for the frequency representation lies in that if two

components are found in the same alignment slot a lot of times in the sample, then they are likely to be grammatically similar.

For one alignment alone, possibility is considered. Since components that are found aligned in a slot need not always follow the one-to-one match, the scoring should assign higher value to pairs that are more likely to be grammatically similar. In alignment based approaches, if there is a large shift in the alignment, the situation is then considered to have low possibility. For example, in a slot, two sentences, [A_1 B_1 C_1 D_1 E_1], [A_2 B_2 C_2 D_2 E_2], where A_1, B_1, C_1, D_1, E_1, A_2, B_2, C_2, D_2, E_2 are sentence components. None of the component is a known match. Some possible real alignments can be:

[A_1 B_1 C_1 D_1 E_1]

[A_2 B_2 C_2 D_2 E_2];

[A_1 B_1 C_1 D_1 E_1]

[A_2 B_2 C_2 D_2 E_2];

[A_1 B_1 C_1 D_1 E_1]

[A_2 B_2 C_2 D_2 E_2];

……

The first one is more possible than the second one with the third one the lowest possibility of the three.

We propose the slot alignment scoring scheme to represent the possibility feature in aligned sentences. In a pairwise alignment for two sentences, denote $S_1=\alpha_1\beta_1\gamma_1\beta_2\alpha_2$, $S_2=\alpha_3\beta_1\gamma_2\beta_2\alpha_4$, where β_1, β_2 are equal constituents beside the slot γ_1 and γ_2, with non-zero lengths. Let the length of γ_1 be m, the length of γ_2 be n, the constituent we are considering in γ_1 is at position i, and that in γ_2 is at position j.

Paths in the DP matrix are used for calculation. Here, we only allow right move and down move, since there is no real substitution. If a down move follows a right move instantly or a right move follows a down move instantly, then the corresponding subsentences are considered to be suggested as a potential substitution. For example, γ_1 = *A B C D* and γ_2 = *E F G*, then the slot alignment score for *B* and *G* is defined as the probability they are aligned as a match in a pairwise alignment.

Suppose a pointer starts at position (0, 0), as shown in Figure 6.3a. It randomly moves right or down with equal possibilities. Accordingly, the move means insertion and deletion. When the pointer reaches the position (*m*, *n*), the path is corresponding to a possible alignment. When the pointer makes a right move directly following a down move, or the pointer makes a down move directly following a right move, it is considered to be a match of the corresponding components (Figure 6.3b). Therefore, the possibility that two components are matched is the number of paths where the components are matched into the total number of possible paths.

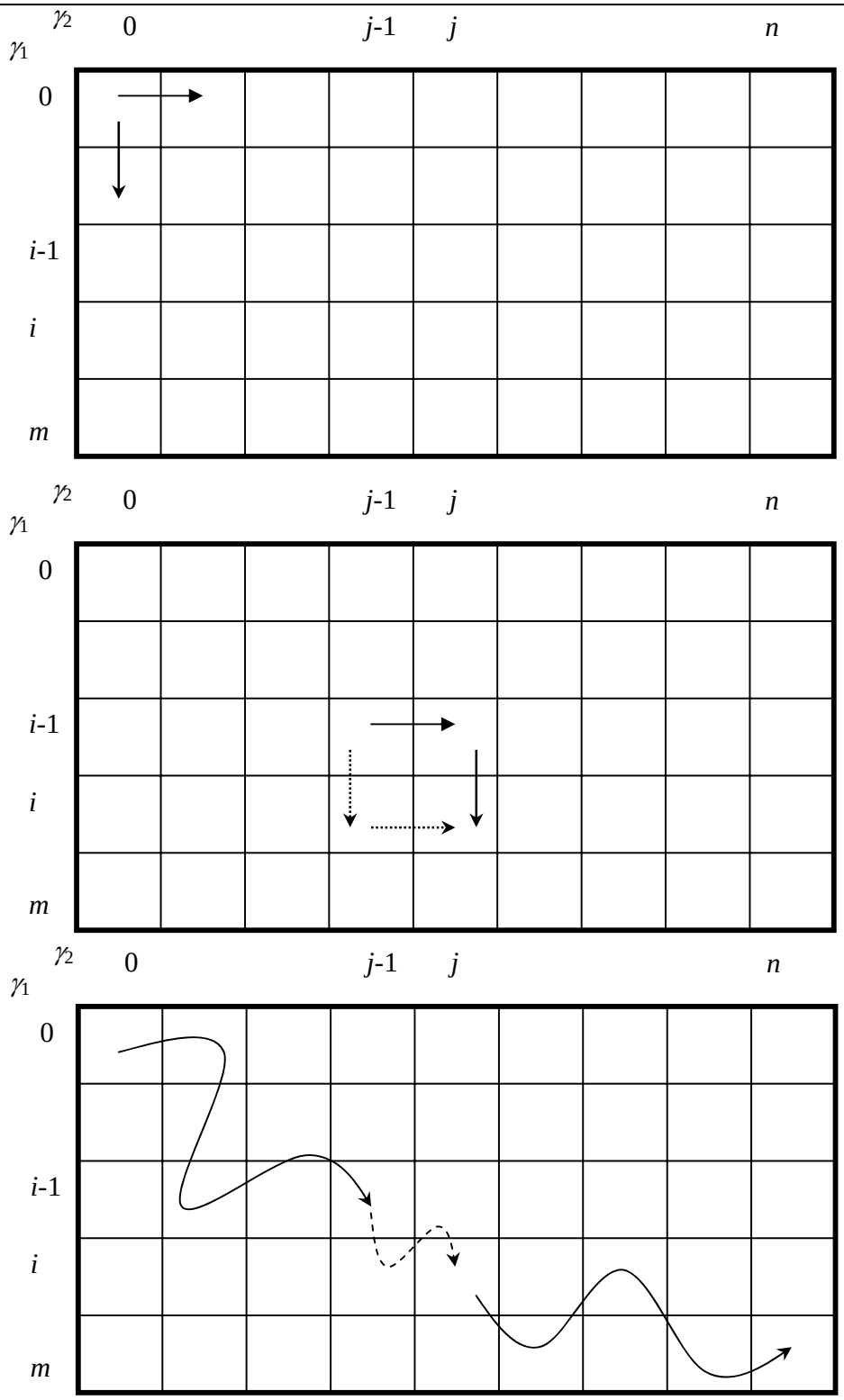


Figure 6.3 Scoring γ1 and γ2 using Alignment Matrix

The number of paths (Figure 6.3c) where the specific components are matched can be calculated by multiplying the number of paths from (0, 0) to (i-1, j-1), the number of paths from (i-1, j-1) to (i, j) and the number of paths from (i, j) to (m, n).

The number of possible paths from (i_1, j_1) to (i_2, j_2) is calculated using the following idea, which is illustrated in Figure 6.4. From (i_1, j_1) to (i_2, j_2), the pointer must move down $(i_2 - i_1)$ times, and move right $(j_2 - j_1)$ times, therefore the number or paths from (i_1, j_1) to (i_2, j_2) is the number of possible selections of $(i_2 - i_1)$ elements from $(i_2 - i_1) + (j_2 - j_1)$ elements, which gives:

$$N[(i_1, j_1) \text{ to } (i_2, j_2)] = \frac{(i_2 - i_1 + j_2 - j_1)!}{(i_2 - i_1)!(j_2 - j_1)!} \tag{6.2}$$

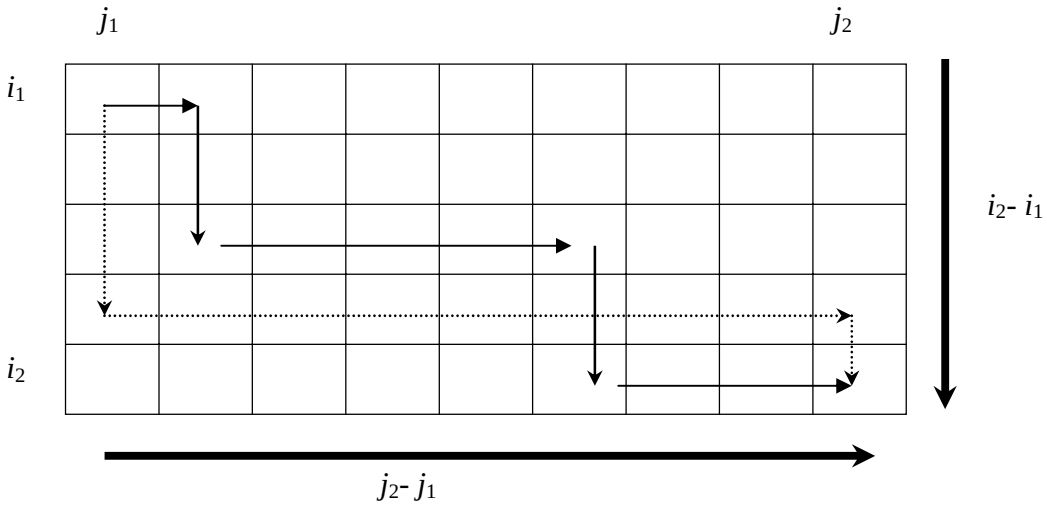


Figure 6.4 Calculating possible paths

Therefore, the total number of paths from (0, 0) to (*m*, *n*) is the number of ways of selecting *m* positions from *m*+*n* positions is given below:

$$N = \frac{(m+n)!}{m!n!} \tag{6.3}$$

Similarly, the number of paths from (0, 0) to (*i*-1, *j*-1) is:

$$N(prefix) = \frac{(i-1+j-1)!}{(i-1)!(j-1)!} \tag{6.4}$$

and the number of paths from (*i*, *j*) to (*m*, *n*) is:

$$N(suffix) = \frac{(m-i+n-j)!}{(m-i)!(n-j)!} \tag{6.5}$$

There are two paths (right then down and down then right) from (*i*-1, *j*-1) to (*i*, *j*). Therefore, the probability of aligning the *i*-th constituent of γ_1 and is the *j*-th constituent of γ_2 is :

$$P(i, j) = \frac{2N(prefix)N(suffix)}{N} \tag{6.6}$$

The $P(i, j)$ is then used as the slot alignment score of the two constituents in the two sentences. The slot alignment scores are accumulated in each pairwise alignment. For stochastic context-free grammars, the accumulated scores in two sentences with counts of $C(p_1)$ and $C(p_2)$ are the slot alignment scores multiply by $C(p_1) \times C(p_2)$.

(3) Alignment Profile

During the alignment phase, subsentences that are aligned in the same slot may be different structural components. In other words, they are at the exact same grammatical context, but not always interchangeable in the sample. For example, in the following sentence pair:

Jane eats apples.

Jane eats well.

by using only pairwise alignment, "*apples*" and '*well*' are considered to form grammatical rules. Context-free grammar rules are generated so that "*apples*" and "*well*" are derivable from the same nonterminal. Therefore the two words are considered to be interchangeable in the sample. However, it is desired that the two constituents could be recognized as different structural components.

We use *Alignment Profile* to capture this difference. The idea behind Alignment profile is illustrated in the next example. Suppose the production rules of the generating context free grammar are:

$S \rightarrow$ *A eats B*

$S \rightarrow$ *A eats C*

$S \rightarrow$ *The job is done C*

$S \rightarrow$ *I brought some B this morning*

$A \rightarrow$ *Jack*

$A \rightarrow$ *Jane*

$B \rightarrow$ *apples*

$B \rightarrow$ *pears*

$C \rightarrow$ *well*

$C \rightarrow$ *badly.*

During alignment phase, sentences generated by *S* are pairwisely used. After that, one may consider using the frequencies of constituents paired with each other as a clue for finding similar constituents, since constituents that are derivable from the same nonterminal have a better chance of being paired with each other than average. However, the number of sentences using the rule $B \rightarrow$ *apples* and those using $B \rightarrow$ *pears* can be quite different. Suppose the frequency of using the word "*apples*" in the sample is much larger than that of using the word "*pears*". Therefore, the word "*well*" must be found paired much more times with the word "*apples*" than with the word "*pears*". If one uses the summed frequencies that constituents are paired with each other as a similarity measure, then the word "*apples*" is considered to be more grammatically similar to the word "*well*" than to the word "*pears*". Analysis reveals that if words are

generated from the same nonterminal, they tend to have similar percentage of frequencies being paired with the words. Suppose we find that the word "*apples*" 4 of 10 times is paired with the word "*well*", 2 of 10 times paired with the word "badly", 3 of 10 times paired with the word "*apples*" and 1 of 10 times paired with the word "*pears*". The word "*pears*" must have similar percentage of frequencies being paired with the words. That is to say, the words with similar percentage of pairing frequencies are very likely to be derived from the same nonterminal.

In the SCFG, there exist productions with the starting symbol *S* as their left hand sides. The productions are same all the way except one nonterminal. To illustrate this, the following two productions are as described.

p_1: $S \rightarrow \alpha_1 A$ ($pr(p_1)$)

p_2: $S \rightarrow \alpha_1 B$ ($pr(p_2)$)

The nonterminals *A* and *B* derive different strings (illustrated as in the following productions).

p_3: $A \rightarrow w_1$ ($pr(p_3)$)

p_4: $A \rightarrow w_2$ ($pr(p_4)$)

p_5: $B \rightarrow v_1$ ($pr(p_5)$)

p_6: $B \rightarrow v_2$ ($pr(p_6)$)

where w_1, w_2, v_1, v_2 are different symbols. The non-terminals A and B are not always interchangeable in the grammar. In other words, there must be some productions like the following:

p_7: $S \rightarrow \alpha_2 A$ ($pr(p_7)$)

p_8: $S \rightarrow \alpha_3 B$ ($pr(p_8)$),

and there is no production of the following type:

p_9: $S \rightarrow \alpha_2 B$ ($pr(p_9)$)

p_{10}: $S \rightarrow \alpha_3 A$ ($pr(p_{10})$),

where α_1 α_2 α_3 are different strings. Therefore, the problem that we need to address is the following. In the alignment learning framework, from sample texts generated by productions p_1 and p_2, the nonterminals A and B are considered interchangeable and hence create rules like:

$C \rightarrow A$,

$C \rightarrow B$,

with the newly created nonterminal C replacing all appearances in the production set.

Some nonterminals are interchangeable in all contexts; however, some (other) nonterminals are not always interchangeable, but they are only interchangeable in the same context. The problem is to identify and handle such nonterminals. We define the

nonterminals that are not always interchangeable to be *distinguishable symbols*; otherwise they are referred to as *indistinguishable symbols*.

Definition 6.1

In a CFG $G(N, \Sigma, P, S)$, $A \in N$, $B \in N$, if for all $p \in P$, RHS(p)=$\alpha A \beta$, there exists a $q \in P$, RHS(q)=$\alpha B \beta$, and LHS(p) = LHS(q), then A and B are called *indistinguishable symbols*, and A is called an *indistinguishable symbol* of B. Otherwise, A and B are *distinguishable symbols*, and A is called a *distinguishable symbol* of B.

Suppose in the previous grammar, if p_9 and p_{10} are in the production set, then A and B are indistinguishable for that they are interchangeable.

Now, consider the pairwise alignment on the sample texts generated by the production set. Suppose enough texts are generated, hence the sample is representative of the production set. Suppose there are $N(S)$ total sentences generated, then the numbers of times that the strings are aligned as a pair are shown below. Denote $N(w, v)$ the number of times that the two symbols w and v are aligned as a pair and $N(w)$ the number of times the symbol w is generated. The numbers that all kinds of sentences are generated are as shown in Figure 6.5.

For $N(w_1, w_2)$, there are only two situations that w_1 and w_2 are aligned as a pair. One is

that both sentences are generated from S using p_1 with one sentence using p_3 and the other using p_4. The other is that both sentences are generated from S using p_7 with one sentence using p_3 and the other using p_4. As shown in Figure 6.5, there are $N(S)\times pr(p_1)\times pr(p_3)$ sentences in the form of $\alpha_1 w_1$ and $N(S)\times pr(p_1)\times pr(p_4)$ sentences in the form of $\alpha_1 w_2$. Therefore, the total number of times that β_1 and β_2 are aligned as a pair, when the sentence is generated by p_1, should be $N(S)\times pr(p_1)\times pr(p_3)\times N(S)\times pr(p_1)\times pr(p_4)$. Similarly, the total number of times that w_1 and w_2 are aligned as a pair, when the sentence is generated by p_7, should be $N(S)\times pr(p_7)\times pr(p_3)\times N(S)\times pr(p_7)\times pr(p_4)$. Hence, we have:

$$N(w_1, w_2)=N(S)\times pr(p_1)\times pr(p_3)\times N(S)\times pr(p_1)\times pr(p_4)+N(S)\times pr(p_7)\times pr(p_3)\times N(S)\times pr(p_7)\times pr(p_4).$$

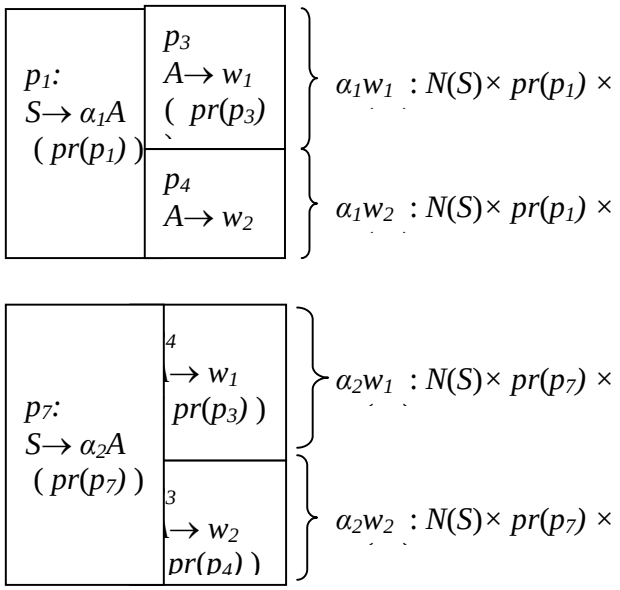


Figure 6.5 Numbers of sentences for calculating $N(w_1, w_2)$

In a similar way, other expressions are calculated as follows.

$N(w_1, w_1) = N(S)\times pr(p_1) \times pr(p_3)\times N(S)\times pr(p_1)\times pr(p_3) + N(S)\times pr(p_7) \times pr(p_3) \times N(S)\times pr(p_7) \times pr(p_3)$

$N(w_1, w_2) = N(S)\times pr(p_1) \times pr(p_3) \times N(S)\times pr(p_1) \times pr(p_4) + N(S)\times pr(p_7) \times pr(p_3) \times N(S)\times pr(p_7) \times pr(p_4)$

$N(w_1, v_1) = N(S)\times pr(p_1) \times pr(p_3) \times N(S)\times pr(p_2) \times pr(p_5)$

$N(w_1, v_2) = N(S)\times pr(p_1) \times pr(p_3) \times N(S)\times pr(p_2) \times pr(p_6)$

$N(w_2, w_1) = N(S)\times pr(p_1) \times pr(p_4) \times N(S)\times pr(p_1) \times pr(p_3) + N(S)\times pr(p_7) \times pr(p_4) \times N(S)\times pr(p_7) \times pr(p_3)$

$N(w_2, w_2) = N(S)\times pr(p_1) \times pr(p_4) \times N(S)\times pr(p_1) \times pr(p_4) + N(S)\times pr(p_7) \times pr(p_4) \times N(S)\times pr(p_7) \times pr(p_4)$

$N(w_2, v_1) = N(S)\times pr(p_1) \times pr(p_4) \times N(S)\times pr(p_2) \times pr(p_5)$

$N(w_2, v_2) = N(S)\times pr(p_1) \times pr(p_4) \times N(S)\times pr(p_2) \times pr(p_6)$

$N(v_1, w_1) = N(S)\times pr(p_2) \times pr(p_5) \times N(S)\times pr(p_1) \times pr(p_3)$

$N(v_1, w_2) = N(S)\times pr(p_2) \times pr(p_5) \times N(S)\times pr(p_1) \times pr(p_4)$

$N(v_1, v_1) = N(S)\times pr(p_2) \times pr(p_5) \times N(S)\times pr(p_2) \times pr(p_5) + N(S)\times pr(p_8) \times pr(p_5) \times N(S)\times pr(p_8) \times pr(p_5)$

$N(v_1, v_2) = N(S)\times pr(p_2) \times pr(p_5) \times N(S)\times pr(p_2) \times pr(p_6) + N(S)\times pr(p_8) \times pr(p_5) \times N(S)\times pr(p_8) \times pr(p_6)$

$N(v_2, w_1) = N(S)\times pr(p_2) \times pr(p_6) \times N(S)\times pr(p_1) \times pr(p_3)$

$N(v_2, w_2) = N(S)\times pr(p_2) \times pr(p_6) \times N(S)\times pr(p_1) \times pr(p_4)$

$N(v_2, v_1) = N(S)\times pr(p_2) \times pr(p_6) \times N(S)\times pr(p_2) \times pr(p_5) + N(S)\times pr(p_8) \times pr(p_6) \times N(S)\times pr(p_8) \times pr(p_5)$

$N(v_2, v_2) = N(S)\times pr(p_2) \times pr(p_6) \times N(S)\times pr(p_2) \times pr(p_6) + N(S)\times pr(p_8) \times pr(p_6) \times N(S)\times pr(p_8) \times pr(p_6)$

From the above, we observe the following:

$N(w_1, w_1) : N(w_2, w_1) = N(w_1, w_2) : N(w_2, w_2)$

$= N(w_1, v_1) : N(w_2, v_1) = N(w_1, v_2) : N(w_2, v_2),$

and,

$N(v_1, w_1) : N(v_2, w_1) = N(v_1, w_2) : N(v_2, w_2)$

$= N(v_1, v_1) : N(v_2, v_1) = N(v_1, v_2) : N(v_2, v_2).$

The same result does not hold between w_1, w_1 and v_1, v_2. Based on the above observation, we propose alignment profile similarity measure based on fuzzy concepts [Setnes 1998, Shen 1993, Zadeh 1965, Zeng 1996] for identifying similar constituents.

Definition 6.2

Based on an SCFG $G(N, \Sigma, P, S)$, an *alignment profile* is a fuzzy set AP' define on the set of symbols $N \cup \Sigma$, denoted $AP' = \{<v, \mu_{AP'}(v)> | v \in N \cup \Sigma\}$, where $\mu_{AP'}(v)$: $N \cup \Sigma \rightarrow [0,1]$ is the membership function of the fuzzy set AP'. The *alignment profile of a symbol* $w \in N \cup \Sigma$ is denoted $AP'(w)$. The membership function of $AP'(w)$ is defined as:

for all $v \in N \cup \Sigma$,

$$\mu_{AP'(w)}(v) = \frac{N(w,v)}{N(w)}, \text{ when } N(w) > 0;$$

$$= 0, \text{ otherwise (when } N(w) = 0).$$

Figure 6.6 illustrates the alignment profile of a symbol w_{k1}.

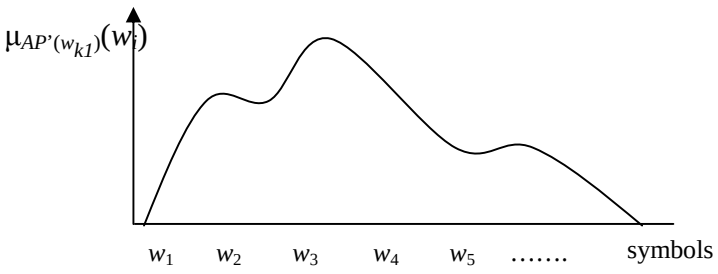


Figure 6.6 Example of Alignment Profile of Symbol w_{k1}

Definition 6.3

Cardinality of an alignment profile AP' is:

$$|AP' = \sum_{v \in N \cup \Sigma} \mu_{AP'}(v).$$

Definition 6.4

The intersection of alignments profiles AP'_1 and AP'_2, denoted as $AP'_1 \cap AP'_2$, is defined by:

$\mu_{AP'_1 \cap AP'_2}(v) = \min(\mu_{AP'_1}(v), \mu_{AP'_2}(v))$, for all $v \in N \cup \Sigma$.

Definition 6.5

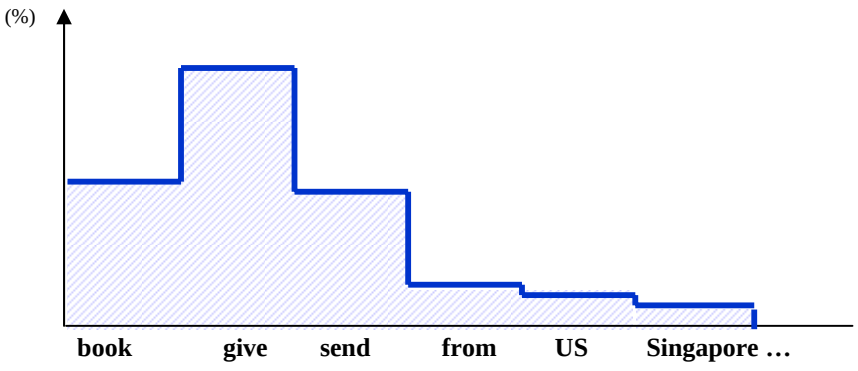
The union of alignments profiles AP'_1 and AP'_2, denoted as $AP'_1 \cup AP'_2$, is defined by:

$\mu_{AP'_1 \cup AP'_2}(v) = \max(\mu_{AP'_1}(v), \mu_{AP'_2}(v))$, for all $v \in N \cup \Sigma$.

Definition 6.6

The *alignment profile similarity* of alignments profiles AP'_1 and AP'_2, denoted as $Ps(AP'_1, AP'_2)$, is defined by:

$$Ps(AP'_1, AP'_2) = \frac{|AP'_1 \cap AP'_2|}{|AP'_1 \cup AP'_2|}.$$



Alignment profile of 'book'

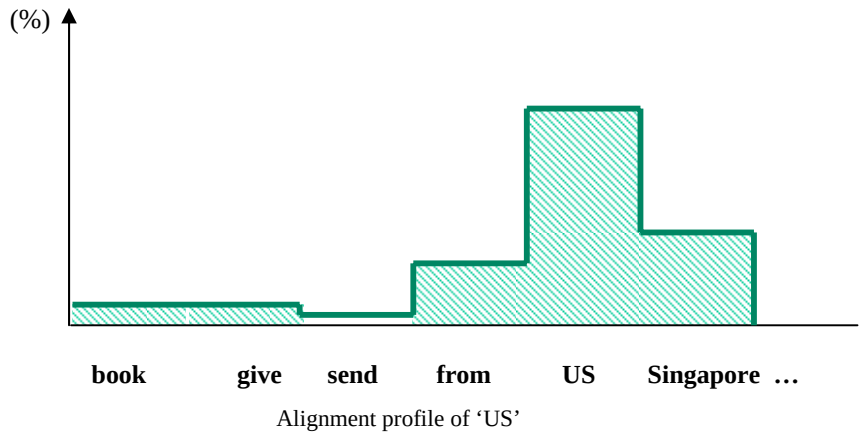


Figure 6.7 Examples of Alignment Profiles

In one iteration of pairwise alignment between sentences, constituents are found paired. Therefore, for each constituent (or non-terminals in the productions), there is a vector of frequencies that a constituents aligned in a pair with all the constituents. We normalize this vector by dividing each component by the sum of all the components, hence the sum of all elements of the normalized vector is one. Such a vector represents alignment 'behavior' of the related constituent, and is called *Alignment Profile*. Figure 6.7 illustrates alignment profiles for words in an air ticket booking application. Figure 6.8 shows how the alignment profiles are used for structural similarity measure, we formalize the method as follows.

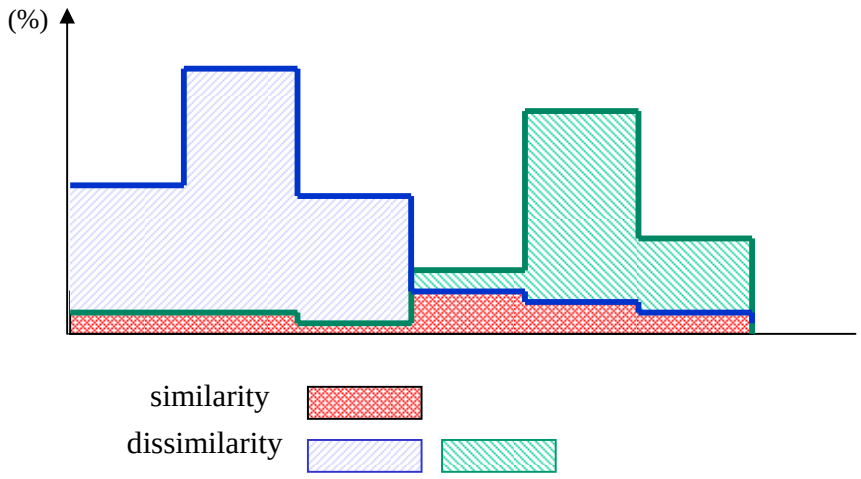


Figure 6.8 Measuring Similarity between Alignment Profiles

To obtain the count of the number of occurrences of the two constituents being paired with each other, we include the alignment of a production rule with itself. If stochastic context-free grammar is processed, each time two constituents are found aligned as pair, suppose the counts of the sentences containing w_i and w_j are $C(p_1)$ and $C(p_2)$, then $N(w_i, w_j)$ is increased by $C(p_1)\times C(p_2)$. When $C(p_1) = C(p_2) = 1$, $N(w_i, w_j)$ is increased by 1.

We have the following theorem as formal statements of the observations.

Theorem 6.1

In a SCFG $G(N, \Sigma, P, S)$, $A\in N$, $w_1\in N\cup\Sigma$, $w_2\in N\cup\Sigma$, $\{A\rightarrow w_1(pr_1),\ A\rightarrow w_2(pr_2)\} \subseteq P$, $pr_1>0$, $pr_2>0$. A can be generated from S in valid sentence. Given enough generated sentences, $Ps(AP'(w_1), AP'(w_2)) = 1$.

PROOF Let $v_1, v_2, \ldots, v_{n-2} \in N\cup\Sigma$ be the other symbols than w_1 and w_2. Let n be

the number of symbols, m be the total number of times when A is generated. Denote $N(A, v_i)$ the number of times when A is pairing with v_i, i = 1,2…n-2. For the learning sample, when we have enough sentences,

$AP(w_1)$

$$= \frac{N(w_1, w_1)}{N(w_1)}/w_1 + \frac{N(w_1, w_2)}{N(w_1)}/w_2 + \frac{N(w_1, v_1)}{N(w_1)}/v_1 + \frac{N(w_1, v_2)}{N(w_1)}/v_2 + \ldots + \frac{N(w_1, v_{n-2})}{N(w_1)}/v_{n-2}$$

$$= m \times pr_1/w_1 + m \times pr_2/w_2 + N(A, v_1)/v_1 + N(A, v_2)/v_2 + \ldots + N(A, v_{n-2})/v_{n-2}.$$

$AP(w_2)$

$$= \frac{N(w_2, w_1)}{N(w_2)}/w_1 + \frac{N(w_2, w_2)}{N(w_2)}/w_2 + \frac{N(w_2, v_1)}{N(w_2)}/v_1 + \frac{N(w_2, v_2)}{N(w_2)}/v_2 + \ldots + \frac{N(w_2, v_{n-2})}{N(w_2)}/v_{n-2}$$

$$= m \times pr_1/w_1 + m \times pr_2/w_2 + N(A, v_1)/v_1 + N(A, v_2)/v_2 + \ldots + N(A, v_{n-2})/v_{n-2}.$$

Since A can be generated from S in valid sentence, m>0, and pr_1>0, pr_2>0, hence, $Ps(AP'(w_1), AP'(w_2))\neq 0$, By definition 6.6, $Ps(AP'(w_1), AP'(w_2)) = 1$.

Q.E.D.

Theorem 6.2

In a SCFG G(N, Σ, P, S), A∈N, B∈N, $w_1 \in N \cup \Sigma$, $w_2 \in N \cup \Sigma$, { $A \rightarrow w_1 (pr_1)$, $B \rightarrow w_2 (pr_2)$ } $\subseteq P$, pr_1>0, pr_2>0. A and B are distinguishable symbols, then the profile similarity $Ps(AP'(w_1), AP'(w_2)) = 1-\sigma$, where σ is a positive real number less than 1.

PROOF Since A and B are distinguishable, there $\exists\, p \in P$, such that:

(i) RHS(p)=$\alpha A\beta$, and there is no $q \in P$, RHS(q)=$\alpha B\beta$, and LHS(p) = LHS(q), or

(ii) there $\exists\ p \in P$, such that RHS(p)=$\alpha B\beta$, and there is no $q \in P$, RHS(q)=$\alpha A\beta$, and LHS(p) = LHS(q).

In case (i) above, we have,

$$\frac{N(w_1, w_1)}{N(w_1)} > \frac{N(w_2, w_1)}{N(w_2)};$$

In case (ii) above, we have,

$$\frac{N(w_2, w_2)}{N(w_2)} > \frac{N(w_1, w_2)}{N(w_1)};$$

When both cases (i) and (ii) occur, we have

$$\frac{N(w_1, w_1)}{N(w_1)} > \frac{N(w_2, w_1)}{N(w_2)} \text{ and } \frac{N(w_2, w_2)}{N(w_2)} > \frac{N(w_1, w_2)}{N(w_1)}.$$

By definition 6.6, $Ps(AP'(w_1), AP'(w_2))<1$. Therefore, $Ps(AP'(w_1), AP'(w_2)) = 1-\sigma$.

Q.E.D.

Theorem 6.1 and 6.2 prove that the alignment profile similarity works as an indicator for specific symbols generated from the same nonterminal. The dissimilarity indicator σ is larger when the alignment profiles of the two symbols are more different.

Slot alignment score can be used to calculate alignment profile. Instead of using the frequencies that two constituents are aligned as a pair, slot alignment scores are used. By definition 6.6, when the slot is a one-to-one match, the incremented slot alignment score is one (or $C(p_1)\times C(p_2)$, in stochastic cases), which is the same with the

incremented amount when the pairing frequencies are used. Additionally, slot alignment score recognized the pairs that are aligned in non-one-to-one slots. Using slot alignment score with alignment profile is proved to be better than using alignment profile (frequencies based only) in experiments shown in Section 6.3.

6.2.4 Repeat Conditions

After the alignment profile similarities are calculated, the substitution matrix is updated accordingly, *i.e.*, alignment profile similarities are set as new corresponding elements of the substitution matrix. If there is no further change to the alignments, then the grammar production rules are updated, or else based on the updated substitution matrix, alignment phase and calculation of the alignment profile similarities which are repeated.

There is an overlapping problem originally stated by Zaanen [Zaanen 2002a]. The problem can be illustrated by the following sentences.

a. [*Oscar sees*] *the apple*.

*b*1. [*Big Bird throws*] *the apple*.

*b*2. *Big Bird* [*throws the apple*] .

c. *Big Bird* [*walks*] .

In the pairwise alignments with the sentence *a* and *c*, sentence *b* is assigned two different alignments *b*1 and *b*2 with overlapping slots. As the production rules

suggested by the two alignments are mutually exclusive, selection must be done among all the overlapping slots. In our work, we use the same method *leaf* presented in [Zaanen 2002a], which select between overlapping identified slots.

The grammar is updated in the following way. In the converged alignments, constituents that are aligned in the same slot are considered to be derived from a new nonterminal. Hence their appearances in the productions involved in the alignments are replaced by a newly created nonterminal. If the replacement results in two exact same productions, one production is deleted. Two additional production rules are created with their left hand side be the new nonterminal and their right hand side be the constituents in the two slots respectively. In stochastic cases, suppose the production counts of the two production rules are $C(p_1)$ and $C(p_2)$. Then the production counts of the newly created production rules (with the left hand side be the new nonterminal) are set to $C(p_1)$ and $C(p_2)$ respectively. If the replacement results in two exact same productions, then one of them is deleted, and the production count of the left one is set to $C(p_1) + C(p_2)$.

The steps described in Section 6.2.2 and 6.2.3 are repeated until there is no further change to the grammar, *i.e.*, no more change to the grammar production rules. For stochastic context-free grammar, the probabilities of each production are calculated by (6.1). Finally, the learned grammar is given as output.

The time complexity of each pairwise alignment is $O(l^2)$, where l is the average length

of each sentence. There are totally $O(m^2)$ such pairwise alignments. Therefore, the complexity of alignment phase is $O(m^2 l^2)$. Suppose n is the input length, $ml = n$, then the complexity of alignment is $O(n^2)$. Then in the profile similarity comparison stage, in the worst case where all the words are different, there are $O(n^2)$ pairs of constituents, and each comparison takes $O(n)$, so the profile calculation is $O(n^3)$. Empirically the number of repeating times k is $O(lg(n))$. Therefore, the complexity of Dynamic SST PBAL is $O(lg(n)\, n^3)$.

6.3 Experimental Results

The data set we used is from CHILDES database which is a collection of corpora of child-related speech [Bliss 1988, Carterette 1974, Chang 1998, Jones 1963, MacWhinney 2000]. We use sample from the English-American corpora, which is about the acquisition of English as a first language in the United States. The number of sentences is over ten thousand; the average sentence length in words is 3.63. The plain texts are exacted as the input. No edition or clean up is done to the sample. Since the systems identify subsentences through alignment, the created rules indicate interchangeabilities between the aligned pair. If the identified pair is interchangeable, *i.e.*, the new sentences created by interchanging the pair are valid, we call that pair is correctly identified, otherwise incorrectly identified. For example, suppose the resultant alignment is like:

[*I*] *am* [*leading*] .

are [*winning*] .

The words "*leading*" and "*winning*" is correctly identified, since after interchanging them, the new sentences

I am winning.

You are leading.

are valid sentences. The words "*I*" and "*you*" are not interchangeable, and therefore the pair is incorrectly identified. The precision is calculated as the percentage of correctly identified pairs.

Figure 6.9 shows comparison results for SST, alignment profile similarity and slot alignment techniques. The settings of the learning procedures are as follows. In *alignment based learning with sentence similarity threshold* (ABL SST), plain Alignment Based Learning [Zaanen 2002a](ABL) runs under a given sentence similarity threshold (SST). Notice that when SST = 0, the method is equivalent to plain ABL. The other two methods are under PBAL framework. In *PBAL not using slot alignment* (PBAL NSA), only alignment profiles are used in updating the substitution matrix. Multiple constituents in the same slots are simply ignored. Each one-to-one match or exact match adds its score to the corresponding slot alignment score, which is used for calculating the alignment profile similarity. In *PBAL using slot alignment* (PBAL SA), everything is exactly as same as in PBAL NSA, except that slot alignment score are calculated used in getting the alignment profiles. The preprocessing, repeat

conditions and other settings in PBAL SA and PBAL NSA follow the PBAL framework. Different SSTs are used in PBAL SA and PBAL NSA. ABST is set to 1, which means not using similar words as slot borders.

Figure 6.10 shows the precision curves where ABST are 1 and 0 respectively. The other settings are as same as those of PBAL SA in the previous experiment. We can see that around the peak area, when ABST = 0 the performance is better than that when ABST = 1. More ABST are tested from 0.5 to 1, with the other setting the same and SST = 4.3. The result is shown in Figure 6.11. It can be seen that as ABST increases, the precision increases. From the results we can see that to get better performance, ABST should be set to 1, *i.e.*, slot border should only be the exact matching words.

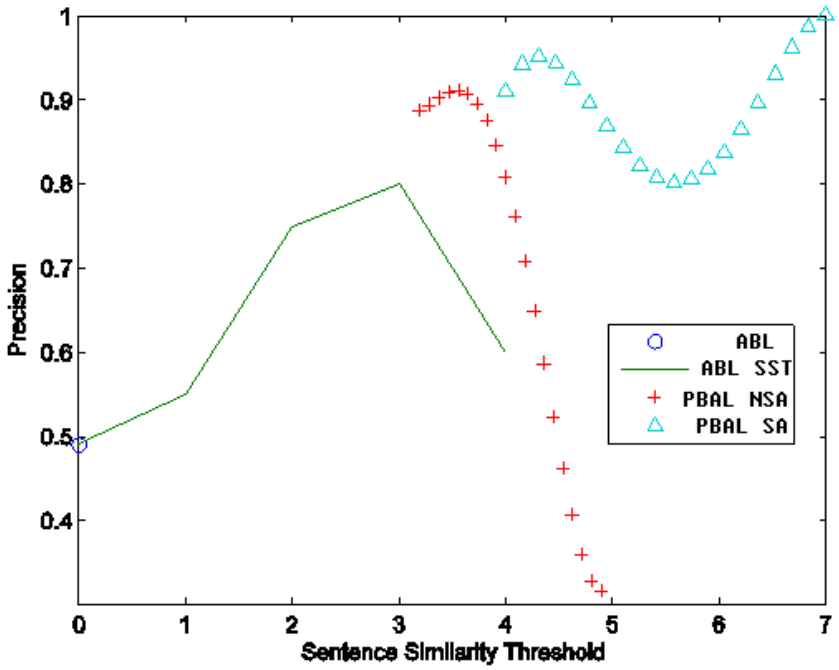


Figure 6.9 Precisions by ABL, ABL SST, PBAL NSA and PBAL SA

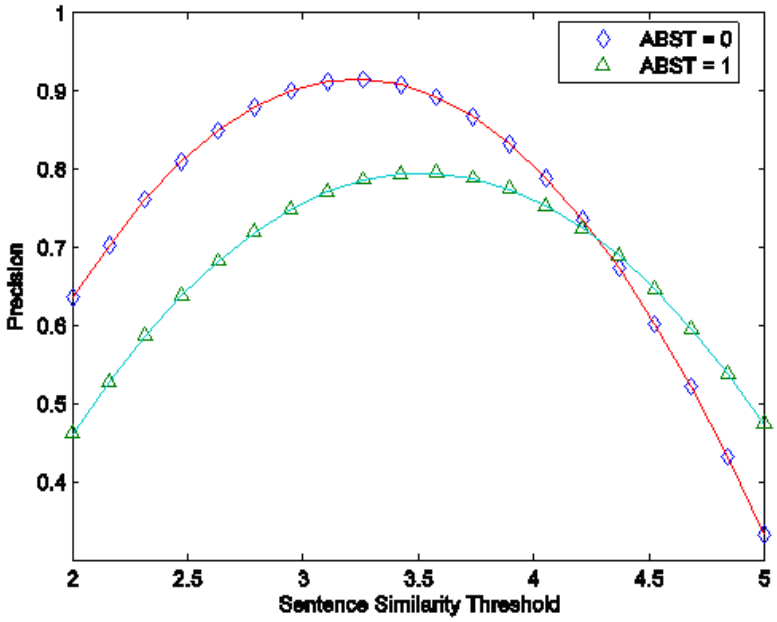


Figure 6.10 Precision curves for ABST 1 and 0.

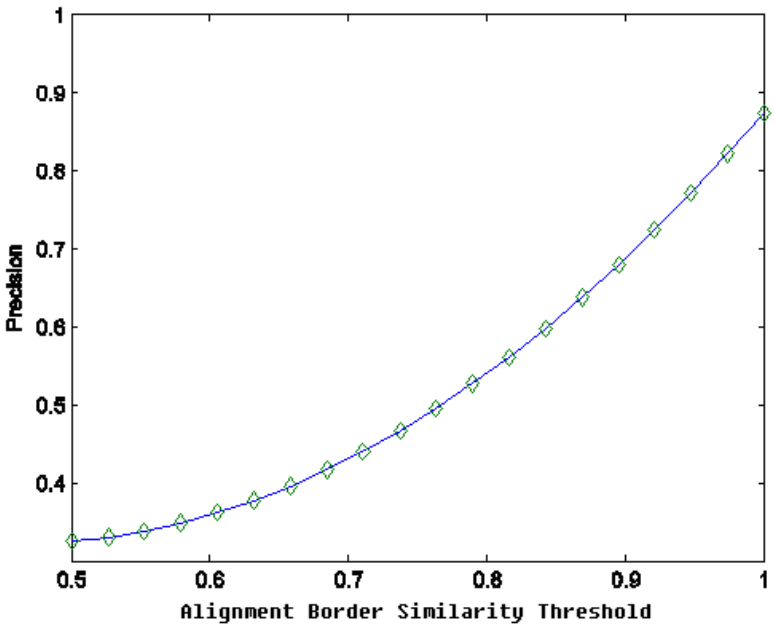


Figure 6.11 Peak Precisions at Different Alignment Border Similarity Thresholds

For *A_Score* based method and *R_Score* based method, since the sentence similarity thresholds are not in the same scale, logarithm (with base 10) of the number of identified patterns are used. The other settings are as same as those of PBAL SA used in the previous experiments. Figure 6.12 shows the comparison between *A_Score* based method and *R_Score* based method. It can be seen that *R_Score* based system identifies more patterns with higher precision.

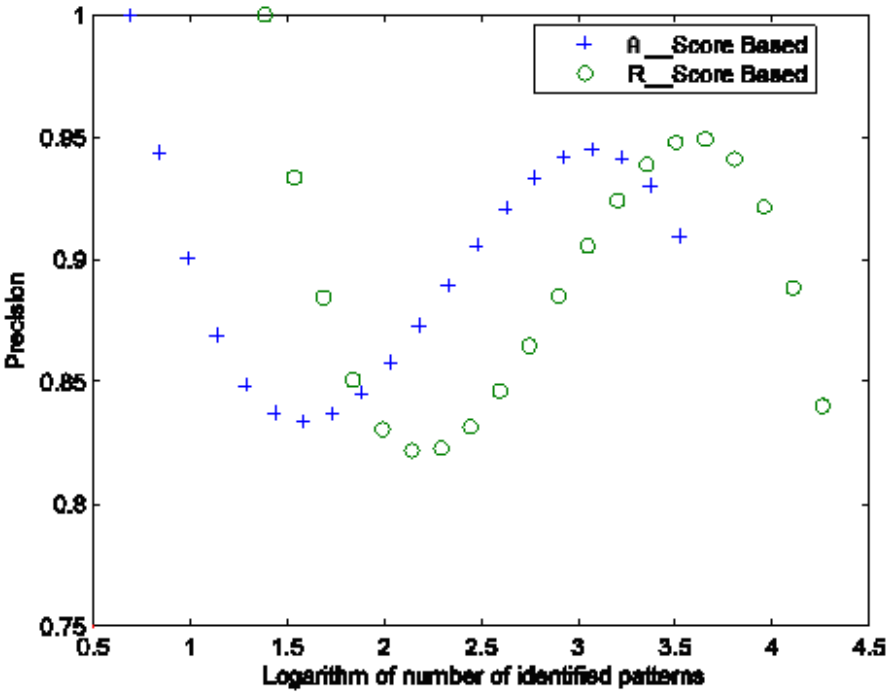


Figure 6.12 Comparing A_Score based system and R_Score based system

Table 6.1 compares the Dynamic SST PBAL to the other methods. For the other methods, peak precisions are used. It can be seen that Dynamic SST PBAL performs the best among the PBAL systems, and it can dynamically find SST and therefore requires no preset value. Figure 6.13 shows three curves of thresholds that are used in iterations. Though the thresholds go up and down through iterations, they usually start highly and drop low when converged.

Table 6.1 Precisions and recall by different PBAL systems

Systems	Precision	Recall
ABL SST	86%	4%
PBAL NSA	78%	6%
PBAL SA	91%	10%
R_Score based PBAL	95%	17%
Dynamic SST PBAL	97%	28%

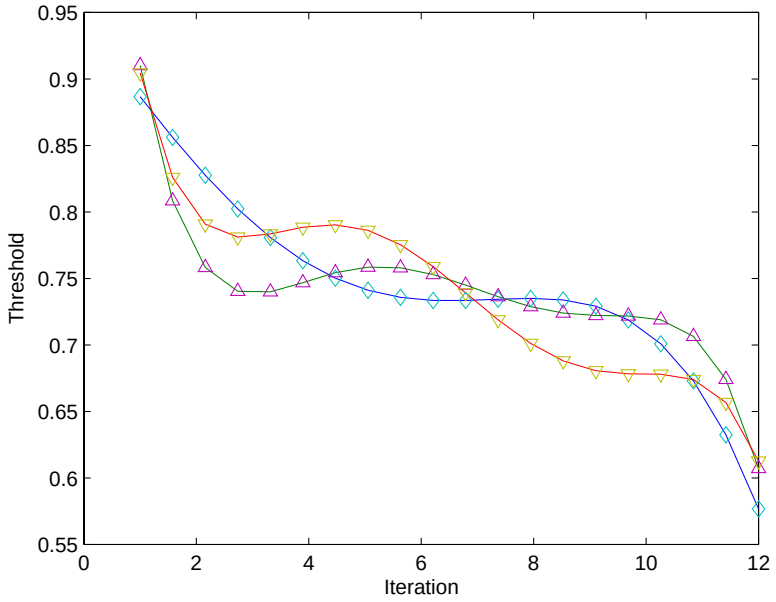


Figure 6.13 Threshold Wave in Dynamic SST PBAL

6.4 Comparison with ABL

While Alignment Based Learning depends only on probabilistic information to select rules after all the hypotheses have been guessed, PBAL systems start from the first set of identified hypotheses. Based on the hypotheses, PBAL systems calculate statistical information, use it to further identify different constituents with similar profiles and eliminate those hypotheses with non-similar profiles. In this way, the alignment and therefore the hypotheses are refined until there are no more changes. This refinement based on statistical information is the main difference of the two frameworks. As a result, the Dynamic SST PBAL improve a lot on the precision with a slightly reduction on the Recall rate (Table 6.2). Also, the model size in term of grammatical rules learned is reduced to 47% from ABL.

Table 6.2 Comparison of ABL and Dynamic SST PBAL

	Precision	Recall	Model Size	Time Complexity
ABL	50%	31%	100%	$O(n^2)$
Dynamic SST PBAL	97%	28%	47%	$O(log(n)n^3)$

However, there is tradeoff between the refinement of the model and time complexity: Dynamic SST PBAL takes time of $O(log(n)n^3)$ to run, mainly due to the calculation of profile similarities.

6.5 Conclusions

We have introduced the collection of methods based on slot alignment scoring and alignment profile to improve the precision of identified grammatical patterns from given symbolic sample in language sentences. Our approach employs statistical information in alignment to optimize further alignments. Based on the optimized alignment results, grammar patterns are identified. Various settings of the system are discussed and compared in experiments. Our technique achieves the best in the dynamic SST PBAL setting, where slot alignment scoring, alignment profile, *R_Score* based sentence similarity threshold and dynamic SST threshold are used. Our experiments indicate usefulness of the proposed techniques on CHILDES dataset.

CHAPTER

7

Conclusions and Future Work

7.1 Conclusions

This thesis addresses some problems in the area of grammatical inference. Due to the non-learnability of the general learning models, proposed approaches use additional information, or artificial intelligence methods.

Our work began with regular language inference. Classification automata were proposed for modeling automaton acceptor representing multiple classes of strings. By doing this, clearer separation and more concise models have been constructed. It is similar to the learning of human being. We learned from samples and created a model for a concept. When more complicated samples were presented to us, we learn to separate the positive and negative samples in a more detailed way. We designed algorithms to construct classification automata under different settings, by state merging and splitting according to backtrack information identified in three conditions.

The correctness and efficiency of the algorithms is proved in the form of theorems.

DFCA is a special extension of DFA. By a maximum word length counter, the number of states of the automaton acceptor can be reduced. By extending the work of Campeanu [Campeanu 2001], we proposed an algorithm to construct minimal DFCA from strings of a finite language. The algorithm is efficient and has complexity $O(n^2)$, where n is the length of the input. This result is proved by theorems and experiments.

Markov chains can be regarded as a special kind of stochastic DFA. The application of Markov chain is very wide. A mixed k-th order Markov chain can represent the samples more accurately than a fixed and ordered one. We presented a method to learn k^{th} order Markov chains by training recurrent neural networks through the given sample sets using a constructive method with time complexity $O(k\ n\ |\Sigma|^{(k+1)})$. By using recurrent neural networks model, we avoided the negative results caused by traditional representations of automata. In this way, the process is able to generate the learning results and can easily extract k^{th} order Markov chains from the resulting recurrent neural networks. Again, the improved accuracy can be seen in the experimental results.

For context-free language learning, we attempted to extract useful grammatical rules rather than learning the exact grammar rules from the sample. We have presented two methods, namely slot alignment scoring and alignment profile, which successfully improve the precision of identified structural patterns from given symbolic samples over alignment based learning. Our approach employs statistical information in

alignment to optimize further alignments. Based on the optimized alignment results, structural patterns are identified. Various settings of the system are discussed and compared in experiments. The system achieves the best in the dynamic SST PBAL setting, where slot alignment scoring, alignment profile, *R_Score* based sentence similarity threshold and dynamic SST threshold are used. The system is demonstrated to identifying more structural patterns with higher precision.

7.2 Future Work

In this section, several possible extensions of the works presented in this thesis are discussed.

7.2.1 Lower bound studies

Methods that refine the learning model sizes are presented in this thesis. We have shown their results by experiments. For grammatical inference, it is very important to study the lower bound of the model size for each learning method.

Kolmogorov complexity [Li 1997] is a potential direction to look at, as this direction reveals the minimal necessary representing cost and also offers methods to compress strings.

7.2.2 Time complexity of PBAL

We have shown that PBAL refines the learning model of ABL at the cost of more time complexity. A possible extension to improve the performance is to use only a constant length for alignment profiles. In the current framework, the length of the alignment profile of a word is the number of all words. The profile vector can be sparse when most of them are 0 since they are unrelated. Therefore using a constant length of vector for alignment profile that only includes the highest scores could be a way to improve the time complexity from the current $O(log(n)n^3)$ to $O(log(n)n^2)$. Since information could be lost during this simplification, the balance of precision and performance should be researched.

7.2.3 The Using of Domain Knowledge

PBAL is a general purpose framework, and does not use the domain knowledge. Problems such as the ones stated in Example 6.7 may be solved by altering the input according to the specific application, such as changing each word into a combination of the original word, the form in use, and its type. This extension will need more research on the properties of the specific domain, such as Natural Language Processing and Bioinformatics.

7.2.4 Association of multiple words

In PBAL, we use identified constituents as slot borders. However, it is more desirable to identify the association of constituents contains of multiple words directly. For example, in the following two sentences:

[*Brand new Zune*] *is on sell at Bellevue Square.*

[*Classical ipods*] *are popular among college students.*

the current PBAL framework depends on identifying the similarity of "is" and "are" to make the correct guess. Although, the 5 words "*Brand*", "*new*", "*Classical*" and "*ipods*" already have similarities above average. It is more desirable to find a way that directly associates such slots in a way similar to sentence similarity.

7.2.5 String Matching Improvements

Efficiency and accuracy are important for practical grammatical inference methods. Real application may have more critical requirement for efficiency. In bioinformatics, DNA sequences are usually of very huge lengths. In web text information applications, the lengths can be large and the number of different words may be quite large. Usually linear algorithms in both time complexity and space complexity are desired. Suffix trees and suffix array are techniques which can improve the running behavior to help finding the patterns in strings, with satisfactory complexity. Data mining techniques due to their ability to handle large volumes of data are also of help in improving the efficiency. Algorithm pruning and heuristic can also help a lot when trivial details can be omitted without harming the accuracy. Therefore, those techniques can be added in our PBAL framework for better practical performance.

Appendix A

List of Publications

The author has contributed a total of 6 international conference publications, 1 book chapter and 1 journal publication. The list is given below.

Conference papers:

- X. Wang and N. S. Chaudhari, "Alignment Based Similarity Measure for Grammar Learning", *2006 IEEE International Conference on Fuzzy Systems*, pp. 1902 – 1909, July 16-21, 2006.

- X. Wang and N. S. Chaudhari, "Profile Based Alignment Learning System for Language Inference", *in Proceedings of the 14th International Conference on Intelligent and Adaptive Systems and Software Engineering*, pp. 94-99, Toronto, Canada, July 20-22, 2005.

- X. Wang and N. S. Chaudhari, "Recurrent Neural Networks For Learning Mixed k^{th} -Order Markov Chains", *in Proceedings of 11th International Conference on Neural Information Processing (ICONIP)*, pp. 477-482, Science City, Calcutta, November 22-25, 2004. Also published in *Lecture Notes in Computer Science* vol 3316, pp. 477-482, 2004.

- X. Wang and N. S. Chaudhari, "Constructing Minimal Cover Automata From Strings", *in Proceedings of Second International Conference on Computational Intelligence, Robotics, and Autonomous Systems (CIRAS)*, Singapore, (Editor,

Prahlad Vadakkepat, Tan Woei Wan, Tan Kay Chen and Loh Ai Poh), Dec 15-18, 2003.

- X. Wang and N. S. Chaudhari, "Classification Automaton and Its Construction Using Learning", *in Proceedings of Sixteenth Canadian Conference on Artificial Intelligence*, Halifax, Nova Scotia, Canada, June 11-13, 2003. Also published in *Advances in Artificial Intelligence: Proceedings - AI 2003, Lecture Notes in Artificial Intelligence (LNAI)* (Y. Xiang, and B. Chaibdraa, Eds.) vol 2671, pp. 515-519, 2003.

- X. Wang and N. S. Chaudhari, An Approach for the Use of Negative Examples for Grammar Learning, in Proceedings of the Second International Conference on Machine Learning and Cybernetics, Xi'an, China, November 2-5 (IEEE Cat. No.03EX693) 3 (2003) 1719-1722.

Book Chapters:

- X. Wang and N. S. Chaudhari, Chapter "Neural Network Based Grammatical Learning and Its Application For Structure Identification", in Neural Networks Applications in Information Technology and Web Engineering, D. Wang, N. K. Lee, Eds. Sarawak, Malaysia: Borneo Publishing, (2005) 15-31.

Journal Papers:

- X. Wang and N. S. Chaudhari, Recurrent Networks for Recognition of Sequence Patterns, International Journal of System Modeling Simulation, Special Issue on Advancement in Algorithms 2 (2004) 1-7.

References

[Abe 1997] N. Abe and H. Mamitsuka, "Predicting protein secondary structure using stochastic tree grammars", *Machine Learning Journal*, 29, pp. 275-310, 1997.

[Adriaans 1992] P. Adriaans, "Language learning from a categorical perspective", Universiteit van Amsterdam, 1992.

[Adriaans 2002] P. Adriaans and M. Vervoort, "The EMILE 4.1 Grammar Induction Toolbox", *in Proceedings of ICGI 2002*, Lecture Notes in Computer Science 2484, pp. 293-295, 2002.

[Angluin 1982] D. Angluin, "Inference of Reversible Languages", *Journal of ACM*, 29, pp. 741-765, 1982.

[Angluin 1987a] D. Angluin, "Learning *k*-bounded Context-free Grammars", *Yale Tech. Rep.*, RR-557, pp., 1987a.

[Angluin 1987b] D. Angluin, "Learning regular sets from queries and counterexamples", *Information and Control*, 39, pp. 337-350, 1987b.

[Angluin 1990] D. Angluin, "Negative results for equivalence queries", *Machine Learning Journal*, 5, pp. 121-150, 1990.

[Angluin 1981] D. Angluin, "A Note on the Number of Queries Needed to Identify Regular Languages", *Information and Control*, 51, pp. 76-87, 1981.

[Angluin 1978] D. Angluin, "On the complexity of minimum inference of regular sets", *Information and Control*, 39, pp. 337-350, 1978.

[Angluin 1988] D. Angluin, "Queries and Concept Learning", *Machine Learning*, 2(4), pp. 319-342, 1988.

[Angluin 1991] D. Angluin and M. Kharitonov, "When won't membership queries help?" *in Proceedings of 24th ACM Symp. on theory of computing*, pp. 444-454, ACM Press, 1991.

[Baker 1979] J. K. Baker, "Trainable Grammars for Speech Recognition", *Speech Communication Papers for the 97th Meeting of the Acoustical Society of America*, pp. 547 -550, 1979.

[BalcRazar 1985] J. L. BalcRazar, J. Diaz and J. GabarrRo, "Uniform characterizations of non-uniform complexity measures", *Information and Control*, 67, pp. 53-89, 1985.

[Bliss 1988] L. Bliss, "The development of modals", *The journal of applied developmental psychology*, 9, pp. 253-261, 1988.

[Bod 2006] R. Bod, "An all-subtrees approach to unsupervised parsing", *in Proceedings of the 21st International Conference on Computational Linguistics and the 44th annual meeting of the ACL*, pp. 865-872, Association for Computational Linguistics, 2006.

[Boden 2000] M. Boden and J. Wiles, "Context-free and context-sensitive dynamics in recurrent neural networks", *Connection Science*, 12(3/4), pp. 197-210, 2000.

[Borges 2000] J. Borges and M. Levene, "Data mining of user navigation patterns", *in Proceedings of Web Usage Mining and User Profiling*, Lecture Notes in Computer Science 1836, pp. 92-111, Springer-Verlag, 2000.

[Brazma 1998] A. Brazma, I. Jonassen, J. Vilo and E. Ukkonen, "Pattern discovery in biosequences", *in Proceedings of ICGI 1998*, Lecture Notes in Artificial Intelligence 1433, pp. 257-270, Springer, Berlin, Heidelberg, 1998.

[Campeanu 2001] C. Campeanu, N. Santean and S. Yu, "Minimal Cover-automata for Finite Languages", *Theoretical Computer Science*, 267, pp. 3-16, 2001.

[Cano 2002] A. Cano, J. Ruiz and P. Garcia, "Inferring Subclasses of Regular Languages Faster Using RPNI and Forbidden Configurations", *in Proceedings of ICGI 2002*, pp. 28-36, Springer, 2002.

[Carterette 1974] E. C. Carterette and M. H. Jones, *Informal speech: alphabetic and phonemic texts with statistical analyses and tables*, University of California Press, Berkeley, CA, 1974.

[Chang 1998] C. Chang, "The development of autonomy in preschool Mandarin Chinese-speaking children's play narratives", *Narrative Inquiry*, 8(1), pp. 77-111, 1998.

[Chidlovskii 2000] B. Chidlovskii, "Wrapper generation by k-reversible grammar induction", *in Proceedings of the Workshop on Machine Learning and Information Extraction*, pp., 2000.

[Cicchello 2002] O. Cicchello and S. C. Kremer, "Beyond EDSM", *in Proceedings of ICGI 2002*, pp. 37-48, Springer, 2002.

[Clark 2001] A. Clark, "Unsupervised induction of stochastic context-free grammars using distributional clustering", *in Proceedings of the 5th Conference on Natural Language Learning*, pp. 1-8, Association for Computational Linguistics, 2001.

[Cook 1976] C. M. Cook, A. Rosenfeld and A. R. Aronson, "Grammatical Inference by Hill-climbing", *Inf. Sciences*, 2(1), pp. 59-80, 1976.

[Crespi-Reghizzi 1971] S. Crespi-Reghizzi, "An Effective Model for Grammar Inference", *Information Processing*, 71, pp. 524-529, 1971.

[Dean 1992] T. Dean, K. Basye, L. Kaelbling, E. Kokkevis, O. Maron, D. Angluin and S. Engelson, "Inferring finite automata with stochastic output functions and an application to map learning", *in Proceedings of the 10th National Conference on Artificial Intelligence*, pp. pp. 208-214, MIT Press, 1992.

[Dupont 1994] P. Dupont, L. Miclet and E. Vidal, "What is the search space of the regular inference?" *in Proceedings of ICGI 1994*, Lecture Notes in Artificial Intelligence 862, pp. 25-37, Springer, Berlin, Heidelberg, 1994.

[Dwork 1990] C. Dwork and L. Stockmeyer, "A time complexity gap for two-way probabilistic finite-state automata", *SIAM J. Comput.*, pp. 1011-1023, 1990.

[Elman 1990] J. Elman, "Finding structure in time", *Cognitive Science*, 14, pp. 179-211, 1990.

[Eren 1997a] H. Eren, C. C. Fung and K. W. Wong, "Application of Artificial Neural Network for prediction of densities and particle size distributions in mineral processing industry", *in Proceedings of IEEE Instrumentation and Measurement Technology*, 2, pp. 1118-1121, 1997a.

[Eren 1997b] H. Eren, C. C. Fung, K. W. Wong and A. Gupta, "Artificial neural networks in estimation of hydrocyclone parameter d50c with unusual input variables", *IEEE Transactions on Instrumentation and Measurement*, 46 (4), pp. 908-912, 1997b.

[Fass 1983] L. F. Fass, "Learning Context-free Languages from Their Structured Sentences", *SIGACT News*, 2(3), pp. 24-35, 1983.

[Garca 1994] P. Garca, E. Segarra, E. Vidal and I. Galian, "On the use of the morphic generator grammatical inference (mggi) methodology in automatic speech recognition", *International Journal of Pattern Recognition and Articial Intelligence*, 4, pp. 667-685, 1994.

[Gers 2001a] F. A. Gers and J. Schmidhuber, "Long short-term memory learns context free and context sensitive languages", *in Proceedings of ICANNGA 2001*, 1, pp. 134-137, Springer, 2001a.

[Gers 2001b] F. A. Gers and J. Schmidhuber, "LSTM recurrent networks learn simple context free and context sensitive languages", *IEEE Transactions on Neural Networks*, 12(6), pp. 1333-1340, 2001b.

[Giles 2001] C. L. Giles, S. Lawrence and A. Tsoi, "Noisy time series prediction using recurrent neural networks and grammatical inference", *Machine Learning Journal*, 44(1), pp. 161-183, 2001.

[Giles 1992] C. L. Giles, C. B. Miller, D. Chen, H. H. Chen, G. Z. Sun and Y. C. Lee, "Learning and extracting finite state automata with second order recurrent networks", *Neural Computation*, 2, pp. 331-402, 1992.

[Gold 1978] E. M. Gold, "Complexity of automaton identification from given data", *Information and Control*, 37, pp. 302-320, 1978.

[Gold 1967] E. M. Gold, "Language Identification in the Limit", *Information and Control*, 10(5), pp. 447-474, 1967.

[Higuera 2005] C. d. l. Higuera, "A Bibliographical Study of Grammatical Inference", *Pattern Recognition*, 38, pp. 1332-1348, 2005.

[Higuera 1996] C. d. l. Higuera, J. Oncina and E. Vidal, "Identification of DFA: data-dependent versus data-independent algorithm", *in Proceedings of ICGI 1996*, pp. 292-300, Springer, 1996.

[Hochreiter 1997] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory", *Neural Computation*, 9(8), pp. 1735-1780, 1997.

[Hopcroft 2001] J. E. Hopcroft, R. Motwani and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison Wesley, Boston, 2001.

[Horning 1972] J. J. Horning, "A Procedure for Grammatical Inference", *Information Processing*, 71, pp. 519-523, 1972.

[Horning 1969] J. J. Horning, "A Study of Grammatical Inference", *Tech. Rep. Computer Science Department, Stanford*, pp., 1969.

[Husken 2003] M. Husken and P. Stagge, "Recurrent neural networks for time series classification", *Neurocomputing*, 50, pp. 223-235, 2003.

[Ishizaka 1990] H. Ishizaka, "Polynomial Time Learnability of Sample Deterministic Languages", *Machine Learning*, 2(2), pp. 151-164, 1990.

[J. T.-L. Wang 1999] J. T.-L. Wang, S. Rozen, B. A. Shapiro, D. Shasha, Z. Wang and M. Yin, "New techniques for DNA sequence classification", *Journal of Computational Biology*, 6(2), pp. 209-218, 1999.

[Jelinek 1998] F. Jelinek, *Statistical Methods for Speech Recognition*, The MIT Press, Cambridge, Massachusetts, 1998.

[Jones 1963] M. H. Jones and E. C. Carterette, "Redundancy in children's free-reading choices", *Journal of verbal learning and verbal behavior*, 2, pp. 489-493, 1963.

[Kaneps 1990] J. Kaneps and R. Frevalds, "Minimal nontrivial space complexity of probabilistics one-way turing machines", *in Proceedings of Mathematical foundations of computer science*, Lecture Notes in Computer Science 452, pp. 355-361, Springer, 1990.

[Kearns 1994] M. Kearns and U. Vazirzni, *An Introduction to Computational Learning Theory*, MIT Press, Cambridge, MA, 1994.

[Klein 2005] D. Klein, "The unsupervised learning of natural language structure", Stanford University, 2005.

[Klein 2001] D. Klein and C. D. Manning, "A generative constituent-context model for improved grammar induction", *in Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, pp. 128-135, Association for Computational Linguistics, 2001.

[Kuhn 2004] J. Kuhn, "Experiments in parallel-text based grammar induction", *in Proceedings of the 42th Annual Meeting on Association for Computational Linguistics*, pp. 470-477, Association for Computational Linguistics, 2004.

[Lang 1992] K. Lang, "Random DFA's can be approximately learned from sparse uniform examples", *in Proceedings of COLT 1992*, pp. 45-52, 1992.

[Lang 1998] K. J. Lang, B. A. Pearlmutter and R. A. Price, "Results of the Abbadingo one DFA learning competition and a new evidence-driven state merging algorithm", *in Proceedings of ICGI '98*, Lecture Notes in Artificial Intelligence 1433, pp. 1-12, Springer-Verlag, 1998.

[Lari 1990] K. Lari and S. J. Young, "The Estimation of Stochastic Context-free Grammars Using the Inside-outside Algorithm", *Comput. Speech Language*, 4, pp. 35-56, 1990.

[Lee 1996] L. Lee, "Learning of Context-Free Languages: A Survey of the Literature", *Tech. Rep. Harvard University*, pp., 1996.

[Levy 1978] L. S. Levy and A. K. Joshi, "Skeletal Structural Descriptions", *Information and Control*, 2(2), pp. 192-211, 1978.

[Li 1997] M. Li and P. Vitanyi, *An Introduction to Kolmogorov Complexity and Its Applications*, 2nd Edition, Springer-Verlag, New York, 1997.

[Luzeaux 1996] D. Luzeaux, "Machine learning applied to the control of complex systems," *in Proceedings of the 8th International Conference on Artificial Intelligence and expert systems applications*, pp., 1996.

[MacWhinney 2000] B. MacWhinney, *The CHILDES Project: Tools for analyzing talk*, third ed., Lawrence Erlbaum Associates, Mahwah, NJ, 2000.

[Makinen 1990] E. Makinen, "The Grammatical Inference Problem for the Szilard Languages of Linear Grammars", *Information Processing Letters*, 2(4), pp. 203-206, 1990.

[McNaughton 1967] R. McNaughton, "Parenthesis Grammars", *Journal. ACM*, 2(3), pp. 490-500, 1967.

[Mohri 1997] M. Mohri, "Finite-state transducers in language and speech processing", *Computational Linguistics*, 23(3), pp. 269-311, 1997.

[Morgan 1995] N. Morgan and H. Bourlard, "An Introduction to Hybrid HMM/Connectionist Continuous Speech Recognition", *IEEE Signal Processing Magazine*, pp. 25-42, 1995.

[Mude 1978] A. V. d. Mude and A. Walker, "On the Inference of Stochastic Regular Grammars", *Information and Control*, 2(5), pp. 310-329, 1978.

[Needleman 1970] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequences of two proteins", *Journal of Molecular Biology*, 48, pp. 443-453, 1970.

[Nevil-Manning 1997] C. Nevil-Manning and I. Witten, "Identifying hierarchical structure in sequences: a linear-time algorithm", *J. Artif. Intell. Res.*, 7, pp. 67-82, 1997.

[Omlin 1996] C. Omlin and C. L. Giles, "Constructing Determinstic Finite State Automata in recurrent neural networks", *Journal of the Association of Computting Machinery (JACM)*, 45, No. 6, pp. 937-972, 1996.

[Oncina 1998] J. Oncina, "The data driven approach applied to the OSTIA algorithm", *in Proceedings of ICGI '98*, Lecture Notes in Artificial Intelligence 1433, pp. 50-56, Springer-Verlag, Berlin, Heidelberg, 1998.

[Oncina 1992] J. Oncina and P. Garcia, *Inferring regular language in polynomial updated time*, World Scientific, 1992.

[Pang 2003] B. Pang, K. Knight and D. Marcu, "Syntax-Based Alignment of Multiple Translations: Extracting Paraphrases and Generating New Sentences", *in Proceedings of Human Language Technology and North American Association for Computational Linguistics Conference*, pp., 2003.

[Parekh 1996] R. J. Parekh and V. Honavar, "An incremental interactive algorithm for regular grammar inference", *in Proceedings of ICGI 1996*, pp. 238-249, Springer, 1996.

[Parekh 1998] R. J. Parekh, C. Nichitiu and V. Honavar, "A polynomial time incremental algorithm for learning DFA", *in Proceedings of ICGI 1998*, Lecture Notes in Artificial Intelligence 1433, pp., Springer-Verlag, Berlin, Heidelberg, 1998.

[Rabiner 1989] L. R. Rabiner, "A tutorial on hidden Markov models and selected applications in speech recognition", *Proc. IEEE*, 77(2), pp. 257-286, 1989.

[Rieger 1995] A. Rieger, "Inferring probabilistic automata from sensor data for robot navigation", *in Proceedings of MLnet Familiarization Workshop and Third European Workshop on Learning Robots*, pp. 64-74, 1995.

[Rodriguez 2001] P. Rodriguez, "Simple recurrent networks learn context-free and context-sensitive language by counting", *Neural Computation*, 13(9), pp. 2093-2118, 2001.

[Sakakibara 1992] Y. Sakakibara, "Efficient Learning of Context-free Grammars from Positive Structural Examples", *Information and Computation*, 97, pp. 23-60, 1992.

[Sakakibara 1990] Y. Sakakibara, "Learning context-free grammars from structural data in polynomial time", *Theoretical Computer Science*, 76, pp. 223-242, 1990.

[Sakakibara 1997] Y. Sakakibara, "Recent Advances of Grammatical Inference", *Theoretical Computer Science*, pp. 15-45, 1997.

[Sakakibara 1994] Y. Sakakibara, M. Brown, R. Hughley, I. Mian, K. Sjolander, R. Underwood and D. Haussler, "Stochastic context-free grammars for tRNA modeling", *Nuclear Acids Res.*, 22, pp. 5112-5120, 1994.

[Salvador 2002] I. Salvador and J. M. Bened, "Rna modeling by combining stochastic context-free grammars and n-gram models", *International Journal of Pattern Recognition and Articial Intelligence*, 16(3), pp. 309-316, 2002.

[Schmidhuber 2002] J. Schmidhuber, F. A. Gers and D. Eck, "Learning Nonregular languages: A Comparison of Simple Recurrent Networks and 'LSTM'", *Neural Computation*, 14(9), pp. 2039-2041, 2002.

[Setnes 1998] M. Setnes, R. Babusaka, U. Kaymak and H. R. v. N. Lemke, "Similarity Measure in Fuzzy Rulebase Simplification", *IEEE Transactions on Systems Man and Cybernetics, Part B: Cybernetics*, 28(3), pp. 376-386, 1998.

[Shallit 1996] J. Shallit and Y. Breitbart, "Automaticity I: properties of a measure of descriptional complexity", *J. Compu. System Sci.*, 53, pp. 10-25, 1996.

[Shen 1993] Q. Shen and R. Leitch, "Fuzzy qualitative simulation", *IEEE Transactions on Systems, Man and Cybernetics*, 23(4), pp. 1038-1061, 1993.

[Solomonoff 1964] R. J. Solomonoff, "A Formal Theory Of Inductive Inference", *Information and Control*, 7, pp. 1-22, 1964.

[Solomonoff 1959] R. J. Solomonoff, "A New Method for Discovering the Grammars of Phrase Structure Languages", *Information Processing*, pp., 1959.

[Takada 1987] Y. Takada, "A Constructive Method for Grammatical Inference of Linear Languages Based on Control Sets", *Tech. Rep. Int. Inst. For Advanced Study in the Soc. Info. Sci.*, pp., 1987.

[Takada 1988] Y. Takada, "Grammatical Inference for Even Linear Languages", *Information Processing Letters*, 28, pp. 193-199, 1988.

[Tarjan 1975] R. E. Tarjan, "Efficiency of a good but not linear set union algorithm", *Journal of ACM*, 22(2), pp. 215-225, 1975.

[Thollard 2000] F. Thollard, P. Dupont and C. d. l. Higuera, "Probabilistic dfa inference using kullback-leibler divergence and minimality", *in Proceedings of 17th International Conf. on Machine Learning*, pp. 975-982, Morgan Kaufmann, San Francisco, CA, 2000.

[Valiant 1984] L. G. Valiant, "A theory of the learnable", *Commun. Assoc. Comput. Mach.*, 27(11), pp. 1134-1142, 1984.

[Vilar 1996] J. M. Vilar, "Query learning of subsequential transducers", *in Proceedings of ICGI '96*, Lecture Notes in Computer Science 1147, pp. 72-83, Springer-Verlag, Berlin, Heidelberg, 1996.

[Wang 2002] Y. Wang and A. Acero, "Grammar learning for spoken language understanding", *in Proceedings of the IEEE Workshop on Automatic Speech Recognition and Understanding*, pp., Madonna di Campiglio, 2002.

[Wharton 1977] R. M. Wharton, "Grammar Enumeration and Inference", *Information and Control*, 2(3), pp. 253-272, 1977.

[Yokomori 1988] T. Yokomori, "Learning Simple Languages in Polynomial Time", *Tech. Rep. SIGFAI, Japanese Society for AI*, pp., 1988.

[Zaanen 2002a] M. V. Zaanen, "Bootstrapping Structure into Language: Alignment-Based Learning", University of Leeds, Leeds, UK, 2002a.

[Zaanen 2002b] M. V. Zaanen, "Implementing Aligment-Based Learning", *in Proceedings of ICGI 2002*, Lecture Notes in Computer Science 2484, pp. 312-314, 2002b.

[Zaanen 2003] M. V. Zaanen, "Theoretical and Practical Experiences with Alignment-Based Learning", *in Proceedings of Australasian Language Technology Workshop (ALTW 2003)*, pp., 2003.

[Zadeh 1965] L. A. Zadeh, "Fuzzy Sets", *Information and Control*, 18, pp. 338-353, 1965.

[Zeng 1996] X.-J. Zeng, Singh, Madan G., "Approximation accuracy analysis of fuzzy systems as function approximators", *IEEE Transactions on Fuzzy Systems*, 4(1), pp. 44-63, 1996.