Theoretical Computational Linguistics: Learning Theory

Jeffrey Heinz

December 19, 2017

Contents

Chapter 1

Analyzing Learning Computationally

1.1 Strings and stringsets

A string is a finite sequence of symbols from some set of symbols Σ. The set of all possible strings is often noted Σ^*. The asterisk is a symbol due to Kleene, who is one of the great computer scientists of the twentieth century. It is often called the 'Kleene star.' Generally, the presence of the Kleene star on a set denotes the **free monoid** of a set, which is the set of all finite sequences of length zero or more from that set. The unique string of length zero is often denoted λ or ϵ.

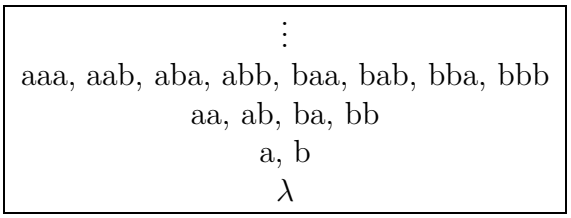


Figure 1.1: Strings of increasing length with $\Sigma = \{a, b\}$.

Here are some examples of sets of strings, also called **formal languages**, or **stringsets**.

1. Let $\Sigma = \{$`a, b, c, ..., z, .`$\}$. Then there is a subset of Σ^* which includes all and only the grammatical sentences of English (modulo capitalization).
2. Let $\Sigma = \{$`Advance-1cm, Turn-R-5°`$\}$. Then there is a subset of Σ^* which includes all and only the ways to get from point A to point B.

Exercise 1. Provide some more examples of stringsets relevant to linguistics.

1.2 The membership problem

The **membership problem** is the problem of deciding whether a string belongs to a set. The problem can be stated thusly: Given a set of strings S and *any* string $s \in \Sigma^*$, output whether $s \in S$. Is there an algorithm that solves this problem for a given S?

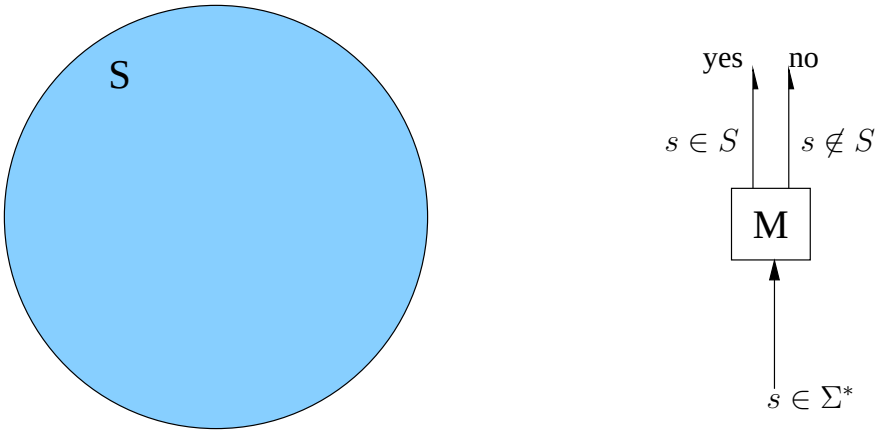
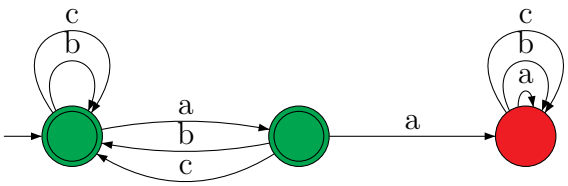


Figure 1.2: The membership problem

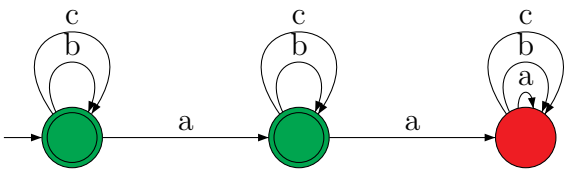
Example 1. A string belongs to S if it does not contain aa as a substring.

$s \in S$	$s \notin S$
abba	baab
abccba	aaccbb
babababa	ccaaccaacc
...	...



Example 2. A string belongs to S if it does not contain aa as a subsequence.

$s \in S$	$s \notin S$
cabb	baab
babccbc	babccba
bbbbbb	bbacccccccccaccc
...	...



Exercise 2. These finite-state machines are not the only algorithmic solutions to these membership problems? Provide other algorithms which solve these two membership problems.

1.3 Learning problems

There are many ways to define the problem of learning a stringset. Here are two informal ones just to get started.

1. For any set S from some given collection of sets: Drawing finitely many examples from S, output a program solving the membership problem for S.

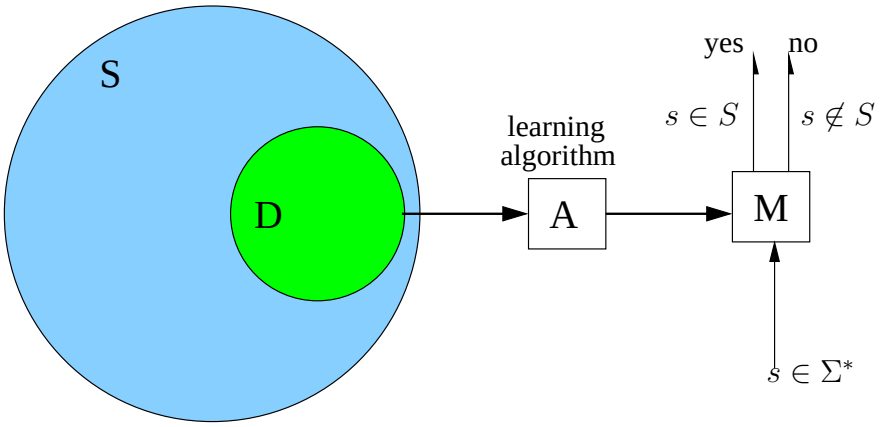


Figure 1.3: A Learning Problem with Only Positive Evidence

2. For any set S from some given collection of sets: Draw finitely many strings labeled as to whether they belong to S or not, output a program solving the membership problem for S.

These definitions are too informal. What does it mean to draw finitely many examples? Do I want the algorithm to succeed for any finite sample of strings providing information about the language? Why or why not?

Exercise 3. Improve the above definitions by making clearer what 'drawing' and 'output' mean. Try to write it formally if you can.

Exercise 4. Recall Osherson *et al.* (1986):

"Of special interest are the circumstances under which these hypotheses stabilize to an accurate representation of the environment from which the evidence is drawn. Such stability and accuracy are conceived as the hallmarks of learning. Within learning theory, the concepts 'evidence,' 'stabilization,' 'accuracy,' and so on, give way to precise definitions."

How do your improved definitions address these concepts?

1.4 Generalizing a little bit

The functional perspective lets us generalize the foregoing a little bit. From a functional perspective a stringset S is associated with a function $f : \Sigma^* \rightarrow \{0, 1\}$. But we may be

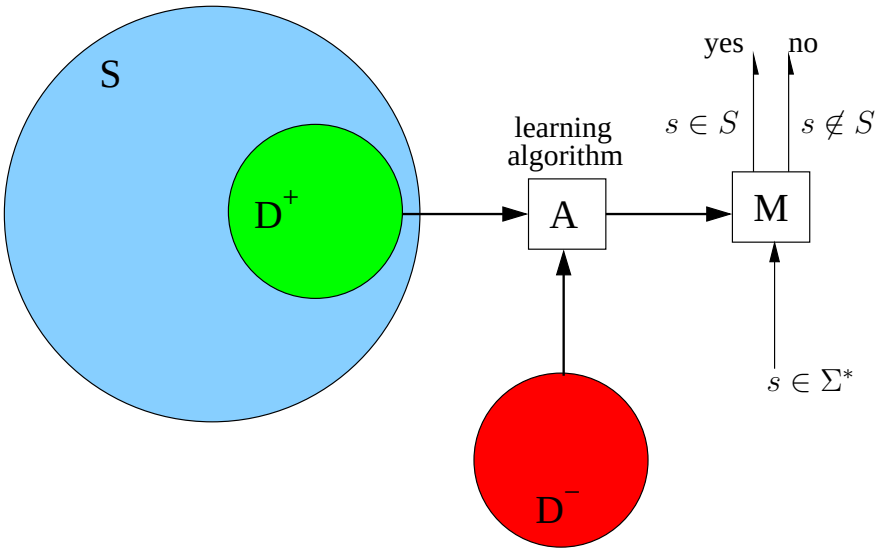


Figure 1.4: Learning Problem with Positive and Negative Evidence

interested in other types of functions which have Σ^* for a domain.

function	Notes
$f : \Sigma^* \to \{0, 1\}$	Binary classification
$f : \Sigma^* \to \mathbb{N}$	Maps strings to numbers
$f : \Sigma^* \to [0, 1]$	Maps strings to real values
$f : \Sigma^* \to \Delta^*$	Maps strings to strings
$f : \Sigma^* \to \wp(\Delta^*)$	Maps strings to sets of strings

Exercise 5. Provide some specific examples of functions like the ones above relevant to linguistics.

Exercise 6. Try to write another definition of learning. It can be for any of the types of functions shown.

1.5 Classifying membership problems

A basic question to ask is whether every stringset has a solution to the membership problem. Perhaps it is surprising to learn that the answer is No. In fact, as a consequence of work on computability in the mid-twentieth century it is known that most stringsets have no solution to the membership problem.

1. Enumerations of Σ^*.
2. No enumeration of $\wp(\Sigma^*)$.

3. Programs are of finite length so programs can be represented as strings of finite length.
4. This means every program is an element of Σ^*.
5. Consequenty there are at most countably many stringsets S which have programs which solve the membership problem of S (see 1).
6. But there are uncountably many stringsets (elements of $\wp(\Sigma^*)$, (see 2)).
7. So most stringsets have no solution to the membership problem.
8. Consequently any conceivable learning problems which targets a a non-enumerable stringset S has no solution because the learning algorithm cannot ultimately output a program which solves the membership problem for S. No such program exists or can exist!

The Chomsky Hierarchy provides additional classification of membership problems which have solutions (i.e. stringsets) (Chomsky, 1956; Hopcroft and Ullman, 1979).

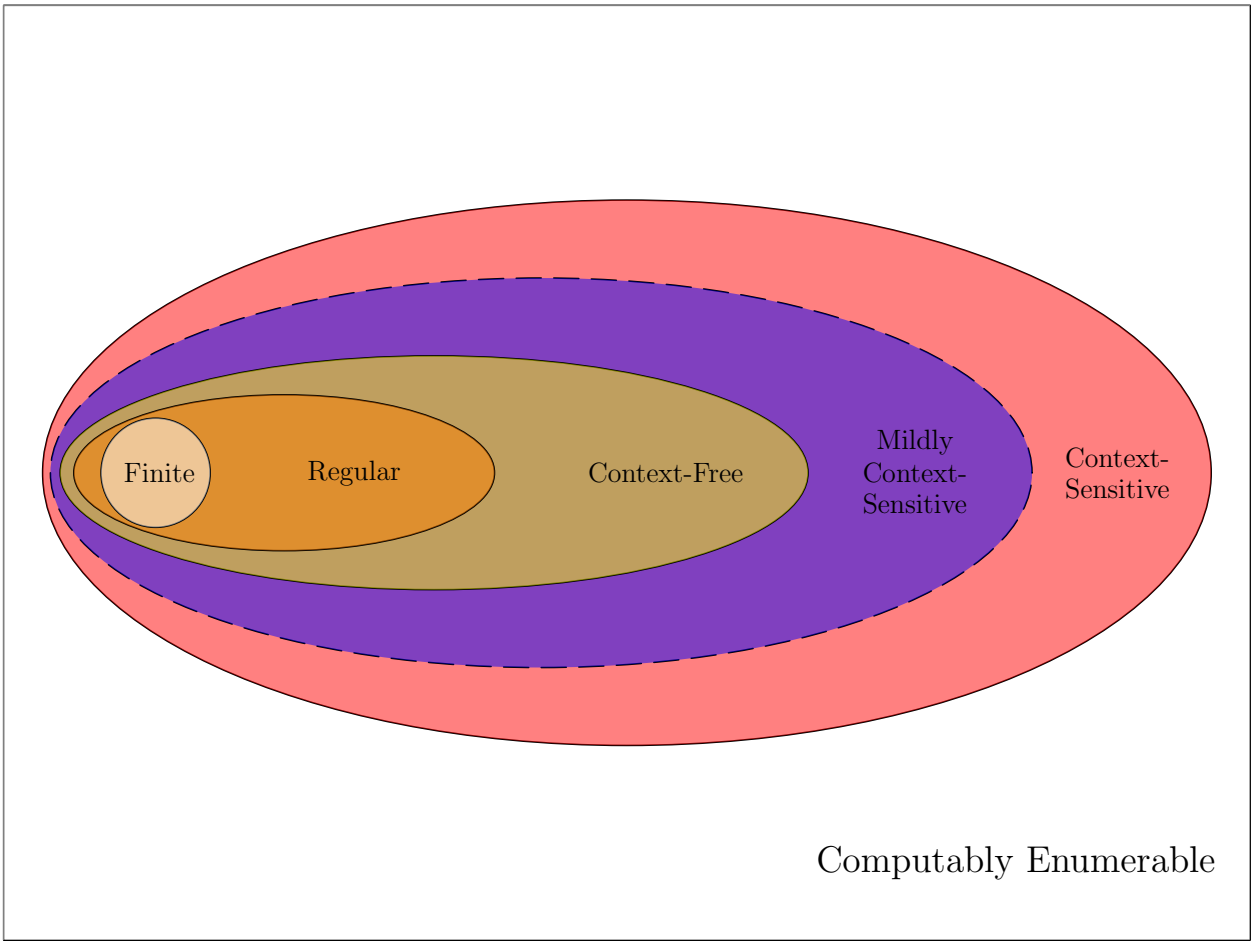


Figure 1.5: The Chomsky Hierarchy

Of particular interest is the class of regular languages, which may be defined as the class of stringsets whose membership problem can be solved by a computational device whose memory requirements are bounded and thus crucically do not grow without bound with

respect to the length of the strings. Finite-state acceptors are one way to represent such computations.

As you may know, zooming in on the class of regular languages reveals some more structure (McNaughton and Papert, 1971; Thomas, 1982).

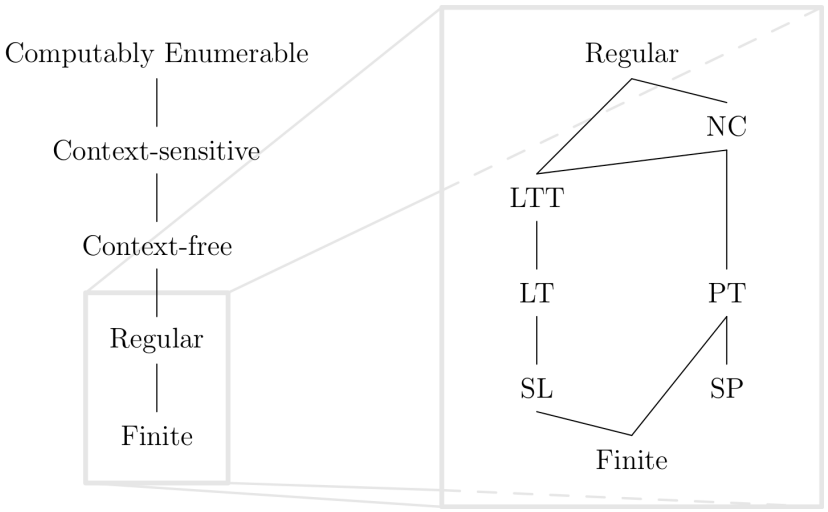


Figure 1.6: Room at the bottom.

These subregular classes are shown below from a model-theoretic perspective (Rogers *et al.*, 2010, 2013).

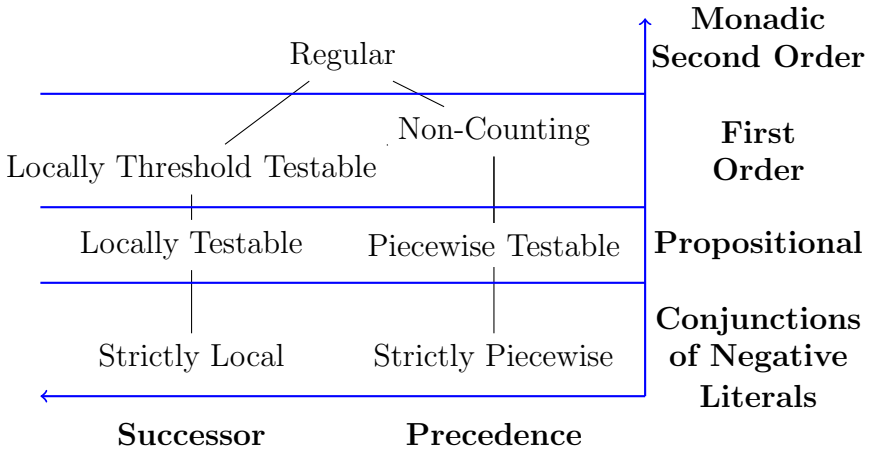


Figure 1.7: Subregular Hierarchies.

Appendix

1.A Enumerating Σ^*

The usual way to enumerate strings in Σ^* is to order them first by their length and then within strings of the same length to order them in dictionary order, as shown below.

0	λ	3	c	6	ac	...
1	a	4	aa	7	ba	
2	b	5	ab	8	bb	

Figure 1.A.1: Enumerating Σ^* with $\Sigma = \{a, b, c\}$.

A natural question that arises is what is the nth string in this enumeration? What effective procedure yields the nth string?

One way to find the nth string is to build a tree of all the strings in a "breadth-first" fashion. The first few steps are shown below.

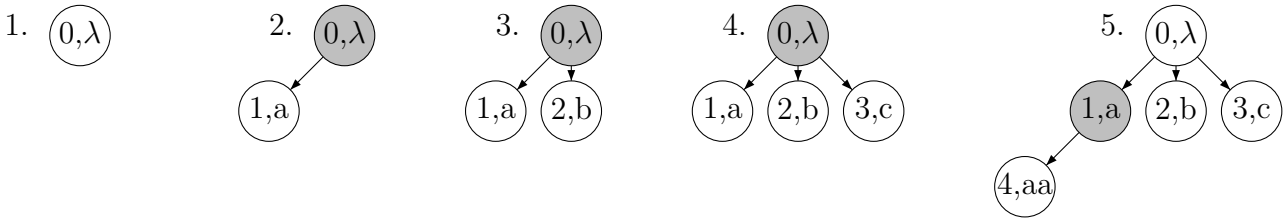


Figure 1.A.2: Enumerating Σ^* with $\Sigma = \{a, b, c\}$.

The procedure for Σ could be stated as follows. Remember we know there are k elements in Σ, and we can assume they are ordered. We are given as input a number n and we want to output the nth string in the enumeration of Σ^*.

1. Set a counter variable c to 0.
2. BUILD a node labeled $(0, \lambda)$.
3. If $0 = n$ then OUTPUT λ and STOP.
4. Otherwise, ADD $(0, \lambda)$ to the QUEUE.
5. REMOVE the first element (m, w) from the QUEUE.
6. Set variable i to 1.
7. Let a be the ith symbol in Σ.
8. Increase c by 1.

9. BUILD a node labeled $(c, w \cdot a)$ as a daughter to (m, w).
10. If $c = n$ then OUTPUT $w \cdot a$ and STOP.
11. Otherwise, ADD this daughter node to the end of the QUEUE.
12. Increase i by 1.
13. If $i > k$ then go to step 5. Otherwise, go to step 7.

The general form of this algorithm is very useful. Recall that an enumeration of Σ^* is also an enumeration of all programs! This means we could try running some set of inputs X on all the programs to find a program that gives a certain output. Basically, in steps 3 and 10 we would check to see how the program w behaves on the inputs in X. If the behavior is what we like, we output this program and stop. Otherwise we continue to the next program!

1.B Non-enumerable stringsets

This is where Alëna Aksenova's handout comes in.

Chapter 2

Identification in the Limit from Positive Data

A definition of a learning problem requires specifying the instances of the problem and specifying what counts as correct answers for these instances. This means thinking carefully about an interaction between three items: the learning targets, the learning algorithm, and the input to the learning algorithm, which can be thought of as the available evidence.

This is difficult because we have to confront the question "Which inputs is it reasonable to expect the learning algorithm to succeed on?" For example, if we are trying to identify a stringset S which is of infinite size but the evidence for S contains only a single string $s \in S$ then we may feel this places an unreasonable burden on the learning algorithm. What is at stake here was expressed by Charles Babbage:

> On two occasions I have been asked [by members of Parliament], "Pray, Mr. Babbage, if you put into the machine wrong figures, will the right answers come out?" I am not able rightly to apprehend the kind of confusion of ideas that could provoke such a question. *as quoted in de la Higuera (2010, p. 391)*

It's unfair to expect a summation algorithm to succeed if the input is wrong. More generally, how do we define learning in such a way so that the input to the algorithm is not "wrong". What does it mean to have input of sufficient quality in learning? We want to only consider instances of the learning problem that are reasonable or fair. But nailing that down precisely is hard! In fact, what we will see is that this is an ongoing issue and there are many attempts to address it. The issue is a live one today.

2.1 Identification in the limit

Gold (1967) provided some influential definitions of learning. He called his approach **identification in the limit**. He provided not one, but several definitions, and he compared what kinds of stringsets were learnable in these paradigms.

No one I know knows what happened to Gold. He seems to have disappeared from academia in the 1980s.

Gold conceptualized learning as a never-ending process unfolding in time. Evidence is presented piece by piece in time to the learning algorithm. The learning algorithm outputs a program with each piece of evidence it receives based on its experience up to the present moment. As time goes on, the programs the learning algorithm outputs must be identical and must solve the membership problem for the target stringset.

Time t	1	2	3	4	...	n	...
Evidence at time t	e(1)	e(2)	e(3)	e(4)	...	e(n)	...
Input to Algorithm at time t	e$\langle 1 \rangle$	e$\langle 2 \rangle$	e$\langle 3 \rangle$	e$\langle 4 \rangle$	...	e$\langle n \rangle$	...
Output of Algorithm at time t	G(1)	G(2)	G(3)	G(4)	...	G(n)	...

Figure 2.1: A schema of the Identification in the Limit learning paradigm

Let us explain the notation in the figure. The notation "e(n)" means the evidence presented at time n. This notation is functional which means evidence can be understood as a function with domain $\mathbb{N}$.

The notation "e$\langle n \rangle$" refers to the sequence of evidence up to the nth one. For example, e(3) means the finite sequence "e(1), e(2), e(3)." In mathematics, angle brackets are sometimes used to denote sequences so some mathematicians would write this sequence as $\langle \, \mathrm{e}(1), \mathrm{e}(2), \mathrm{e}(3) \, \rangle$.

The notation "G(n)" refers to the program output by the algorithm with input e$\langle n \rangle$. If A is the algorithm, and we wish to use functional notation so that $A(i) = o$ means "on input i, algorithm A outputs o" then G(n)=A(e$\langle n \rangle$).

There are two important ideas in this paradigm. First, a successful learning algorithm is one that converges over time to a correct generalization. At some time point n, the algorithm must output the same program and this program must solve the membership problem for S. This means the algorithm can make mistakes, *but only finitely many times.*

Second, which infinite sequences of evidence learners must succeed on? Which are the ones of sufficient quality? Gold defined required these sequences to be representative of the target stringsets. *Each* possible piece of evidence *occurs at some point* in the unfolding sequence of evidence. Lest we think this is too good to be true, recall that the input to the learner at any given point n in time is the *finite* sequence e$\langle n \rangle$, and that to succeed, it is only allowed to make finitely many mistakes.

2.1.1 Definition of identification in the limit from positive data

The box below precisely defines the paradigm when learning from *positive* data. Let us define the "evidence" when learning from positive data more precisely. A **positive presentation** of a stringset S is a function $\varphi : \mathbb{N} \to S$ such that φ is onto. Recall that a function f is

onto provided for every element y in its co-domain there is some element x in its domain such that $f(x) = y$. Here, this means for every string $s \in S$, there is some $n \in \mathbb{N}$ such that $\varphi(n) = s$.

Definition 1 (Identification in the limit from positive data)**.**

1 Algorithm A *identifies in the limit from positive data* a class of stringsets C provided
2 for all stringsets $S \in C$,
3 for all positive presentations φ of S,
4 there is some number $n \in \mathbb{N}$ such that
5 for all $m > n$,
6 • the program output by A on $\varphi\langle m \rangle$ is the same as the the program
7 output by A on $\varphi\langle n \rangle$, and
8 • the program output by A on $\varphi\langle m \rangle$ solves the membership problem
9 for S.

Here is breakdown of what these lines mean.

Line 1 Establishes the name of the relationship between an algorithm A and a collection of stringsets C provided the definition holds.
Line 2 The algorithm must succeed for all $S \in C$.
Line 3 The algorithm must succeed for all positive presentations φ of S.
Line 4 It succeeds on φ for S if there is a point in time n
Line 5 such that for all future points in time m,
Lines 6-7 the output of A converges to the same program, and
Lines 8-9 the output of A correctly solves the membership problem for S.
This paradigm is also called **learning from text.**

2.1.2 The Strictly k-Piecewise Stringsets

Example 3. Here we present an algorithm and prove that it identifies the Strictly k-Piecewise (SP_k) stringsets in the limit from positive data. SP stringsets were proposed to model aspects of long-distance phonotactics Heinz (2010a), motivated on typological and learnability grounds. The learning scheme discussed here exemplifies more general ideas Heinz (2010b); Heinz *et al.* (2012).

The notion of *subsequence* is integral to SP stringsets. Informally, a string u is subsequence of string v if one is left with u after erasing zero or more letters in v. For example, ab is a subsequence of $cccccacccccccccbccccccc$. Formally, u is a subsequence of v ($u \sqsubseteq v$) provided there are strings $x_1, x_2, \ldots x_n$ and strings $y_0, y_1, \ldots y_n$ such that $u = x_1 x_2 \ldots x_n$ and $v = y_0 x_1 y_1 x_2 y_2 \ldots x_n y_n$. It is the y_i strings that erased in v to leave u.

A stringset S is Strictly Piecewise if and only if it is closed under subsequence. In other words, if $s \in S$ then every subsequence of s is also in S.

A theorem shows that every SP stringset S has a basis in a finite set of strings (Rogers *et al.*, 2010). These strings can be understood as *forbidden* subsequences. That is any string

$s \in \Sigma^*$ containing any one of the forbidden subsequences is not in S. Conversely, any string s which does not contain any forbidden subsequence belongs to S.

The same theorem shows that a SP stringset S can be defined in terms of a finite set of *permissible* subsequences. Because the set is finite, there is a longest string in this set. Let its length be k. In this case, any $s \in \Sigma^*$ belongs to S if and only if every one of its subsequences of length k or less is permissible.

In other words we can define SP_k stringsets as follows. Let a grammar G be a finite subset of Σ^* and let k be the length of a longest string in G. Let $\mathtt{subseq}_k(s) = \{u \mid u \sqsubseteq s, |u| \leq k\}$. The "language of the grammar" $L(G)$ is defined as the stringset $\{s \mid \mathtt{subseq}_k(s) \subseteq G\}$. We are going to be interested in the collection of stringsets SP_k, defined as those stringsets generated from grammars G with a longest string k. Formally,

$$\mathrm{SP}_k \stackrel{\text{def}}{=} \{S \mid G \subseteq \Sigma^{\leq k}, L(G) = S\} \ .$$

This is the collection C of learning targets.

For all $S \in \mathrm{SP}_k$, all presentations φ of S, and all time points $t \in \mathbb{N}$ define A as follows:

$$A\big(\varphi\langle t\rangle\big) = \begin{cases} \mathtt{subseq}_k(\varphi(t)) & \text{if } t = 1 \\ A(\varphi\langle t-1\rangle) \cup \mathtt{subseq}_k(\varphi(t)) & \text{otherwise} \end{cases}$$

One can prove that algorithm A identifies in the limit from positive data the collection of stringsets SP_k.

Exercise 7. Prove algorithm A identifies in the limit from positive data the collection of stringsets SP_k.

For any presentation ϕ and time t, define k-SPIA (Strictly k-Piecewise Inference Algorithm) as follows

$$k\text{-SPIA}\big(\varphi\langle t\rangle\big) = \begin{cases} \varnothing & \text{if } t = 0 \\ k\text{-SPIA}(\varphi\langle t-1\rangle) \cup \mathtt{subseq}_k(\varphi(t)) & \text{otherwise} \end{cases}$$

Note that we are being a little sloppy here. Technically, the output of k-SPIA given some input sequence is a set of subsequences G, not a program. What we really mean with the above is that $k-$SPIA outputs a program which uses G to solve the membership problem for $L(G) = \{w \mid \mathtt{subseq}_k(w) \subseteq G\}$. This program looks something like this.

1. Input: any word w.
2. Check whether $\mathtt{subseq}_k(w) \subseteq G$.
3. If so, OUTPUT Yes, otherwise OUTPUT No.

All $k-$SPIA does is update this program simply by updating the contents of G.

Theorem 1. *For each k, $k-$*SPIA *identifies in the limit from positive data the collection of stringsets SP_k.*

Proof Consider any $k \in \mathbb{N}$. Consider any $S \in \mathrm{SP}_k$. Consider any positive presentation φ for S. It is sufficient to show there exists a point in time t_ℓ such that for all $m \geq t_\ell$ the following holds:

1. k-SPIA($\langle m \rangle$) = $k-$SPIA($\langle t_\ell \rangle$) (convergence), and
2. k-SPIA($\langle m \rangle$) is a program that solves the membership probem for S.

Since $S \in \mathrm{SP}_k$, there is a finite set $G \subseteq \Sigma^{\leq k}$ such that $S = L(G)$.

Consider any subsequence $g \in G$. Since $g \in G$ there is some word $w \in S$ which contains g as a k-subsequence. Since G is finite, there are finitely many such w, one for each g in G. Because φ is a positive presentation for S, there is a time t where each of these w occurs. For each w let t be the first occurence of w in φ. Let t_ℓ denote the latest time point of all of these time points t. Next we argue that for all time points m larger than this t_ℓ, the output of $k-$SPIA correctly solves the membership problem for S and does not change.

Consider any $m \geq t_\ell$. The claim is that k-SPIA($\langle m \rangle$) = $k-$SPIA($\langle t_\ell \rangle$) = G. For each g in G, a word containing g as a subsequence occurs at or earlier than t_ℓ and so $g \in k-$SPIA($\langle m \rangle$). Since g was arbitrary in G, $G \subseteq k-$SPIA($\langle m \rangle$).

Similarly, for each $g \in k-$SPIA($\langle m \rangle$), there was some word w in φ such that w contains g as a subsequence. Since φ is a positive presentation for S, w is in S. Since w belongs to S, $\texttt{subseq}_k(w) \subseteq G$ and so g belongs to G. Since g was arbitrary in k-SPIA($\langle m \rangle$) it follows that k-SPIA($\langle m \rangle$) $\subseteq G$.

It follows k-SPIA($\langle m \rangle$) = G.

Since m was arbitrarily larger than t_ℓ we have both convergence and correctness.

Since φ was arbitrary for S, S arbitrary in SP_k and k arbitrary, the proof is concluded. □

2.1.3 The Strictly k-Local Stringsets

Here we present an algorithm and prove that it identifies the Strictly k-Local (SL_k) stringsets in the limit from positive data. The first proof of this result was presented by Garcia *et al.* (1990), though the Markovian principles underlying this result were understood in a statistical context much earlier. The learning scheme discussed there exemplifies more general ideas (Heinz, 2010b; Heinz *et al.*, 2012).

The notion of *substring* is integral to SL stringsets. Formally, a string u is substring of string v ($u \trianglelefteq v$) provided there are strings $x, y \in \Sigma^*$ and $v = xuy$. Another term for substring is *factor*. So we also say that u is a factor of v. If u is of length k then we say u is a k-factor of v.

A stringset S is Strictly k-Local if and only if there is a number k such that for all strings $u_1, v_1, u_2, v_2, x \in \Sigma^*$ such that if $|x| = k$ and $u_1xv_1, u_2xv_2 \in S$ then $u_1xv_2 \in S$. We say S is closed under suffix substitution (Rogers and Pullum, 2011).

A theorem shows that every SL_k stringset S has a basis in a finite set of strings (Rogers and Pullum, 2011). These strings can be understood as *forbidden* substrings. Informally, this means any string s containing any one of the forbidden substrings is not in S. Conversely, any string s which does not contain any forbidden substring belongs to S.

The same theorem shows that a SL stringset S can be defined in terms of a finite set of *permissible* substrings. In this case, s belongs to S if and only if every one of its k-factors is permissible.

We formalize the above notions by first defining a function the $\texttt{factor}_k$, which extracts the substrings of length k present in a string, or those present in a set of strings. If a string s is of length less than k then $\texttt{factor}_k$ just returns s.

Formally, let $\texttt{factor}_k(s)$ equal $\{u \mid u \trianglelefteq s, |u| = k\}$ whenever $k \leq |s|$ and let $\texttt{factor}_k(s) = \{s\}$ whenever $|s| < k$. We expand the domain of this function to include sets of strings as follows: $\texttt{factor}_k(S) = \bigcup_{s \in S} \texttt{factor}_k(s)$.

To formally define SL_k grammars, we introduce the symbols $\rtimes$ and $\ltimes$, which denote left and right word boundaries, respectively. These symbols are introduced because we also want to be able to forbid specific strings at the beginning and ends of words, and traditionally strictly local stringsets were defined to make such distinctions (McNaughton and Papert, 1971). Then let a grammar G be a finite subset of $\texttt{factor}_k(\{\rtimes\}\Sigma^*\{\ltimes\})$.

The "language of the grammar" $L(G)$ is defined as the stringset $\{s \mid \texttt{factor}_k(\rtimes s \ltimes) \subseteq G\}$. We are going to be interested in the collection of stringsets SL_k, defined as those stringsets generated from grammars G with a longest string k. Formally,

$$\mathrm{SL}_k \stackrel{\text{def}}{=} \{S \mid G \subseteq \texttt{factor}_k(\{\rtimes\}\Sigma^*\{\ltimes\}), L(G) = S\} \ .$$

This is the collection C of learning targets.

For all $S \in \mathrm{SL}_k$, for any presentation ϕ and time t, define k-SPIA (Strictly k-Local Inference Algorithm) as follows

$$k\text{-SLIA}\big(\varphi\langle t\rangle\big) = \begin{cases} \varnothing & \text{if } t = 0 \\ k\text{-SLIA}(\varphi\langle t-1\rangle) \cup \texttt{factor}_k(\rtimes\varphi(t)\ltimes) & \text{otherwise} \end{cases}$$

Exercise 8. Prove algorithm k-SLIA identifies in the limit from positive data the collection of stringsets SL_k.

Note that we are being a little sloppy here. Technically, the output of k-SLIA given some input sequence is a set of subsequences G, not a program. What we really mean with the above is that $k-$SLIA outputs a program which uses G to solve the membership problem for $L(G) = \{w \mid \texttt{subseq}_k(w) \subseteq G\}$. This program looks something like this.

1. Input: any word w.
2. Check whether $\texttt{factor}_k(\rtimes w \ltimes) \subseteq G$.
3. If so, OUTPUT Yes, otherwise OUTPUT No.

All $k-$SLIA does is update this program simply by updating the contents of G.

Theorem 2. *For each k, $k-$SLIA identifies in the limit from positive data the collection of stringsets SL_k.*

Proof Consider any $k \in \mathbb{N}$. Consider any $S \in \mathrm{SL}_k$. Consider any positive presentation φ for S. It is sufficient to show there exists a point in time t_ℓ such that for all $m \geq t_\ell$ the following holds:

1. $k\text{-SLIA}(\langle m\rangle) = k-\text{SLIA}(\langle t_\ell\rangle)$ (convergence), and
2. $k\text{-SLIA}(\langle m\rangle)$ is a program that solves the membership probem for S.

Since $S \in \mathrm{SL}_k$, there is a finite set $G \subseteq \Sigma^{\leq k}$ such that $S = L(G)$.

Consider any factor $g \in G$. Since $g \in G$ there is some word $w \in S$ which contains g as a k-factor. Since G is finite, there are finitely many such w, one for each g in G. Because φ is a positive presentation for S, there is a time t where each of these w occurs. For each w let t be the first occurence of w in φ. Let t_ℓ denote the latest time point of all of these time points t. Next we argue that for all time points m larger than this t_ℓ, the output of $k-$SLIA correctly solves the membership problem for S and does not change.

Consider any $m \geq t_\ell$. The claim is that k-SLIA$(\langle m \rangle) = k-$SLIA$(\langle t_\ell \rangle) = G$. For each g in G, a word containing g as a factor occurs at or earlier than t_ℓ and so $g \in k-$SLIA$(\langle m \rangle)$. Since g was arbitrary in G, $G \subseteq k-$SLIA$(\langle m \rangle)$.

Similarly, for each $g \in k-$SLIA$(\langle m \rangle)$, there was some word w in φ such that w contains g as a factor. Since φ is a positive presentation for S, w is in S. Since w belongs to S, $\texttt{factor}_k(w) \subseteq G$ and so g belongs to G. Since g was arbitrary in k-SLIA$(\langle m \rangle)$ it follows that k-SLIA$(\langle m \rangle) \subseteq G$.

It follows k-SLIA$(\langle m \rangle) = G$.

Since m was arbitrarily larger than t_ℓ we have both convergence and correctness.

Since φ was arbitrary for S, S arbitrary in SL_k and k arbitrary, the proof is concluded. □

2.1.4 Strictly k-Local Treesets

Trees are like strings in that they are recursive structures. Informally, trees are structures with a single 'root' which dominates a sequence of trees.

Defining Trees and Treesets

Formally, trees extend the dimensionality of string structures from 1 to 2 (Rogers, 2003). Like strings, we assume a set of symbols Σ. This is often partioned into symbols of different types depending on whether the symbols can only occur at the leaves of the trees or not. We don't make any such distinction here.

Definition 2 (Trees)**.**
Base Case: If $a \in \Sigma$ then $a[\,]$ is a tree.
Inductive Case: If $a \in \Sigma$ and $t_1, t_2, \ldots t_n$ is a string of trees of length n then $a[t_1 t_2 \ldots t_n]$ is a tree.

Also, a tree $a[]$ is called a *leaf*. We denote set of all possible trees with $\mathbb{T}^2_\Sigma$. A *treeset* T is a subset of $\mathbb{T}^2_\Sigma$.

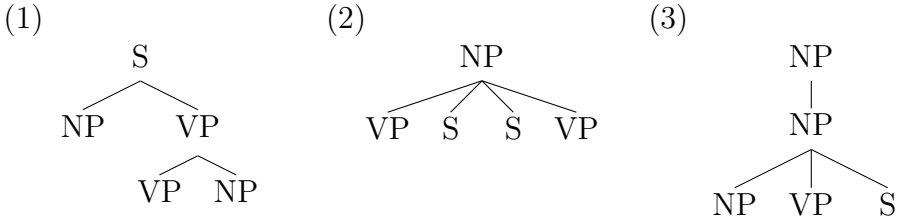
(This notation follows Rogers (2003) wherein $\mathbb{T}^d_\Sigma$ denotes tree-like structures with Σ being the set of labels and d being the dimensionality. Since strings are of dimension 1, this means the set of all strings Σ^* is equivalent to $\mathbb{T}^1_\Sigma$.)

Here are some examples.

Example 4. Let $\Sigma = \{\mathrm{NP}, \mathrm{VP}, \mathrm{S}\}$. Then the following are trees.

1. S[NP[ ] VP[VP[ ] NP[ ]]]
2. NP[VP[ ] S[ ] S[ ] VP[ ]]
3. NP[NP[NP[ ] VP[ ] S[ ]]]

We might draw these structures as follows.



Note that the expression "a string of trees" in the definition of trees implies that our alphabet for strings is all the trees. Since we are defining trees, this may seem a bit circular. The key to resolving this circularity is to interleave the definition of the alphabet of the strings with the definition of trees in a zig-zag fashion. First we apply the inductive case for trees *once*, then we use those trees as an alphabet to define some strings of trees. Then we go back to trees and apply the inductive case again, which yields more trees which we can use to enlarge our set of strings and so on. While we do not go through the details here, this method essentially provides a way to enumerate the set of all possible trees.

Here are some useful definitions which give us information about trees.

Definition 3.

1. The *root* of a tree $a[t_1 \dots t_n]$ is a.
2. The *size* of a tree t, written $|t|$, is defined as follows. If $t = a[\ ]$, its size is 1. If not, then $t = a[t_1 t_2 \dots t_n]$ where each t_i is a tree. Then $|t| = 1 + |t_1| + |t_2| + \dots + |t_n|$.
3. The *depth* of a tree t, written $\texttt{depth}(t)$, is defined as follows. If $t = a[\ ]$, its depth is 1. If not, then $t = a[t_1 t_2 \dots t_n]$ where each t_i is a tree. Then $\texttt{depth}(t) = 1 + \texttt{max}\big\{\texttt{depth}(t_1), \texttt{depth}(t_2), \dots, \texttt{depth}(t_n)\big\}$ where $\texttt{max}$ takes the largest number in the set.
4. The *yield* of a tree t, written $\texttt{yield}(t)$, maps a tree to a string of its leaves as follows. If $t = a[\ ]$ then $\texttt{yield}(t) = a$. If not, then $t = a[t_1 t_2 \dots t_n]$ where each t_i is a tree. Then $\texttt{yield}(t) = \texttt{yield}(t_1) \cdot \texttt{yield}(t_2) \cdot \dots \cdot \texttt{yield}(t_n)$.
5. Tree t is a *subtree* of $t' = a[t_1 \dots t_n]$ provided there is i such that either $t = t_i$ or t is a subtree of t_i.
6. A tree $t = a[a_1[\ ] \dots a_n[\ ]\ ]$ is a *2-local tree* of tree t' ($t \trianglelefteq t'$) provided there exists a subtree s of t' such that $s = a[t_1 \dots t_n]$ and the root of each t_i is a_i.
7. A 1-*treetop* of $t = a[t_1 t_2 \dots t_n]$ is $a[\ ]$.
8. A k-*treetop* of $t = a[t_1 t_2 \dots t_n]$ is $a[s_1 s_2 \dots s_n]$ where each s_i is a $(k-1)$-treetop of t_i.
9. A tree $t = a[t_1 \dots t_n]$ is a k-*local tree* of tree t' ($t \trianglelefteq t'$) provided
   (a) t is of depth k
   (b) there exists a subtree s of t' such that $s = a[s_1 \dots s_n]$ and for each i, t_i is the $(k-1)$ treetop of s_i.

Strictly Local Treesets

The notion of *k-local tree* ($\trianglelefteq$) is integral to Strictly Local treesets.

As with SL stringsets, we define a function $\texttt{factor}_k$, which extracts the k-local trees present in a tree (or set of trees). If a tree t is of depth less than k then $\texttt{factor}_k$ just returns $\{t\}$.

Formally, let $\texttt{factor}_k(t)$ equal $\{s \mid s \trianglelefteq t, \texttt{depth}(s) = k\}$ whenever $k \leq \texttt{depth}(t)$ and let $\texttt{factor}_k(t) = \{t\}$ whenever $\texttt{depth}(t) < k$. We expand the domain of this function to include sets of strings as follows: $\texttt{factor}_k(T) = \bigcup_{t \in T} \texttt{factor}_k(t)$.

Then a SL_k treeset grammar $G = (\Sigma_0, \Sigma_\ell, F_k)$, which will be interpreted as the symbols permissible as the roots, the symbols permissible as the leaves, and the permissible k-Local trees. So $\Sigma_0, \Sigma_\ell \subseteq \Sigma$ and F_k is a finite subset of $\texttt{factor}_k(\mathbb{T}^2_\Sigma)$.

The "language of the grammar" $L(G)$ is defined as the treeset

$$L\big((\Sigma_0, \Sigma_\ell, F_k)\big) \stackrel{\text{def}}{=} \{t \mid \texttt{root}(t) \in \Sigma_0, \texttt{yield}(t) \in \Sigma^*_\ell, \texttt{factor}_k(t) \subseteq F_k, \} \ .$$

We are going to be interested in the collection of treesets SLT_k, defined as those treesets generated from grammars G where the depth of the largest permissible local tree is k. Formally,

$$\text{SLT}_k \stackrel{\text{def}}{=} \{T \mid \exists G = (\Sigma_0, \Sigma_\ell, F_k), \Sigma_0 \subseteq \Sigma,\ \Sigma_\ell \subseteq \Sigma,\ F_k \subseteq \texttt{factor}_k(\mathbb{T}^2_\Sigma),\ F_k \text{ finite},\ L(G) = T\} \ .$$

This is the collection C of learning targets.

Theorem 3. *For each k, the class k-SLT is identifiable in the limit from positive data.*

To make this result clear, let us remind ourselves what a positive presentation of data is. It means for each $t \in T$ there is some time point i such that $\phi(i) = t$. So the evidence here are tree structures, not the yields of tree structures.

Exercise 9. Prove the theorem.

Context-Free Grammars

Definition 4. A *context-free grammar* is a tuple $\langle T, N, S, \mathcal{R}\rangle$ where

- $\mathcal{T}$ is a nonempty finite alphabet of symbols. These symbols are also called the *terminal* symbols, and we usually write them with lowercase letters like $a, b, c, \ldots$
- $\mathcal{N}$ is a nonempty finite set of *non-terminal* symbols, which are distinct from elements of $\mathcal{T}$. These symbols are also called *category* symbols, and we usually write them with uppercase letters like $A, B, C, \ldots$
- S is the *start* category, which is an element of $\mathcal{N}$.
- A finite set of *production rules* $\mathcal{R}$. A production rule has the form

  $$A \rightarrow \beta$$

  where β belongs to $(\mathcal{T} \cup \mathcal{N})^*$ and $A \in \mathcal{N}$. So β are strings of non-terminal and terminal symbols and A is a non-terminal.

Example 5. Consider the following grammar G_1:

- $\mathcal{T} = \{\textbf{\textit{john}}, \textbf{\textit{laughed}}, \textbf{\textit{and}}\}$;
- $\mathcal{N} = \{\text{S, VP1, VP2}\}$; and
- $$\mathcal{R} = \left\{ \begin{array}{l} \text{S} \rightarrow \textbf{\textit{john}}\ \text{VP1} \\ \text{VP1} \rightarrow \textbf{\textit{laughed}} \\ \text{VP1} \rightarrow \textbf{\textit{laughed}}\ \text{VP2} \\ \text{VP2} \rightarrow \textbf{\textit{and laughed}}\ \text{VP2} \\ \text{VP2} \rightarrow \textbf{\textit{laughed}} \end{array} \right\}$$

Example 6. Consider the following grammar G_2:

- $T = \{\text{a, b}\}$;
- $V = \{\text{S}\}$; and
- The production rules are
$$\mathcal{R} = \left\{ \begin{array}{l} \text{S} \rightarrow \text{aSb} \\ \text{S} \rightarrow ab \end{array} \right\}$$

The language of a context-free grammar (CFG) is defined recursively below.

Definition 5. The (partial) *derivations* of a CFG $G = \langle \mathcal{T}, \mathcal{N}, S, \mathcal{R} \rangle$ is written $D(G)$ and is defined recursively as follows.

1. *The base case:* S belongs to $D(G)$.
2. *The recursive case:* For all $A \rightarrow \beta \in \mathcal{R}$ and for all $\gamma_1, \gamma_2 \in (\mathcal{T} \cup \mathcal{N})^*$, if $\gamma_1 A \gamma_2 \in D(G)$ then $\gamma_1 \beta \gamma_2 \in D(G)$.
3. Nothing else is in $D(G)$.

Then the language of the grammar $L(G)$ is defined as

$$L(G) = \big\{ w \in \mathcal{T}^* \mid w \in D(G) \big\}.$$

Exercise 10. How does G_1 generate ***John laughed and laughed and laughed***?

Exercise 11. What language does G_2 generate?

Theorem 4. *The languages generated by context-free grammars are exactly the yields of the Strictly 2-Local treesets.*

Exercise 12. Explain how the 2-local trees in a tree relate to the production rules of a context-free grammar.

Chapter 3

Identification in the Limit: General Results

In this chapter review general results in the identification in the limit paradigm. We begin with some theorems for learning from positive data.

We have already seen that the following classes of languages are identifiable in the limit from positive data.

1. BAR-X $= \{\bar{x} \mid x \in \Sigma^*\}$. Recall $\bar{x} \stackrel{\text{def}}{=} \{w \in \Sigma^* \mid w \neq x\}$.
2. For each $k \in \mathbb{N}$, SP_k
3. For each $k \in \mathbb{N}$, SL_k

The following classes of stringsets are fundamental ones in formal language theory so it makes sense to be curious about their learnability.

1. The class of finite stringsets (FIN).
2. The class of regular stringsets (REG).
3. The class of context-free stringsets (CF).
4. The class of context-sensitive stringsets (CS).

These are in the following relationship: FIN $\subsetneq$ REG $\subsetneq$ CF $\subsetneq$ CF.

3.1 Identification in the limit from positive data

Theorem 5. *FIN is identifiable in the limit from positive data.*

Exercise 13. Prove this theorem. (Hint: FIN can be learned with string extension learning.)

Any class which includes every finite language and at least one more is a *superfinite* class of languages.

Theorem 6. *No superfinite class of stringsets is identifiable in the limit from positive data.*

There are different ways to prove this theorem. Here is one based on (de la Higuera, 2010, p. 151).

Proof[sketch] Consider any superfinite class of languages C. By definition C includes all finite languages and at least one infinite language L_∞.

Let $x_1, x_2, \ldots$ be the infinitely many words of L_∞.

Let $L_1 = \{x_1\}$, $L_2 = L_1 \cup \{x_2\}$, $L_3 = L_2 \cup \{x_3\}$, and so on. So $L_k = L_{k-1} \cup \{x_k\}$. For each $k \in \mathbb{N}$, $L_k \in C$ since L_k is finite.

For the sake of contradiction, assume there is an algorithm A that identifies C in the limit from positive data. We will show there is a presentation for L_∞ for which A fails to converge.

Pick a presentation φ_1 for L_1. Since A identifies L_1 in the limit, there is a convergence point i_1 such that A outputs a grammar for L on $\varphi_1[i_1]$. Let φ_2 be some presentation of L_2 such that for all $j < i_1$, $\varphi_2(j) = \varphi_1(j)$ and $\varphi_2(i_1 + 1) = x_2$. More generally, let φ_k be some presentation of L_k such that for all $j < i_{k-1}$, $\varphi_k(j) = \varphi_{k-1}(j)$ and $\varphi_k(i_{k-1} + 1) = x_k$.

In this manner we construct a presentation φ_∞ for L_∞. Consider any $i \in \mathbb{N}$. There exists $j, j+1$ such that $i_j < i \leq i_{j+1}$. Let k equal $j + 1$. Then $\varphi_\infty(i)$ equals $\varphi_k(i)$.

How does A behave on φ_∞? It does not converge. This is because for all $k \in \mathbb{N}$, at time point i_k, A will output a program for L_k. So it never converges to a grammar for L_∞ even though φ_∞ is a positive presentation for L_∞. □

Gold explains the idea behind his result this way.

> It is of great interest to find why information presentation by text is so weak and under what circumstances it becomes stronger. Therefore, it is worthwhile to understand the method used in Theorems I.8 and I.9 to prove that any class of languages containing all finite languages and at least one infinite language is not identifiable in the limit from a text in five out of six of the models using text.
>
> The basic idea is proof by contradiction. Consider any proposed guessing algorithm. It must identify any finite language correctly after a finite amount of text. This makes it possible to construct a text for the infinite language which will fool the learner into making a wrong guess an infinite number of times as follows. The text ranges over successively larger, finite subsets of the infinite language. At each stage it repeats the elements of the current subset long enough to fool the learner. Thus, the method of proof of the negative results concerning text depends on the possibility of there being a huge amount of repetition in the text. Perhaps this can be prevented by some reasonable probabilistic assumption concerning the generation of the text. In this case one would only require identification in the limit with probability one, rather than for every allowed text.
>
> I have been asked, "If information presentation is by means of text, why not guess the unknown language to be the simplest one which accepts the text available?" This is identification by enumeration. It is instructive to see why it will not work for most interesting classes of languages: The universal language (if it is in the class) will have some finite complexity. If the unknown language is more complex, then the guessing procedure being considered will always guess wrong,

> since the universal language is consistent with any finite text. This follows from the fact that, if L is the unknown language and if L' $\supset$ L, then L' is consistent with any finite segment of any text for L. The problem with text is that, if you guess too large a language, the text will never tell you that you are wrong.

It immediately follows that the class of regular, context-free, context-sensitive, and computably enumerable classes of stringsets are not identifiable in the limit from positive data.

Furthemore, for every finite stringset S, there is some k such that S is Strictly k-Local. Thus FIN $\subsetneq$ SL. Hence neither SL nor LT nor LTT nor TSL is identifiable in the limit from positive data.

Theorem 7 ((Angluin, 1980))**.** *A class C is identifiable in the limit from positive data iff for each $S \in C$ there is a finite set $D \subseteq S$ such that for all $S' \in C$ such that $D \subseteq S'$ it holds that $S' \not\subseteq S$.*

Pictorially, Figure 3.1 is the situation that cannot obtain.

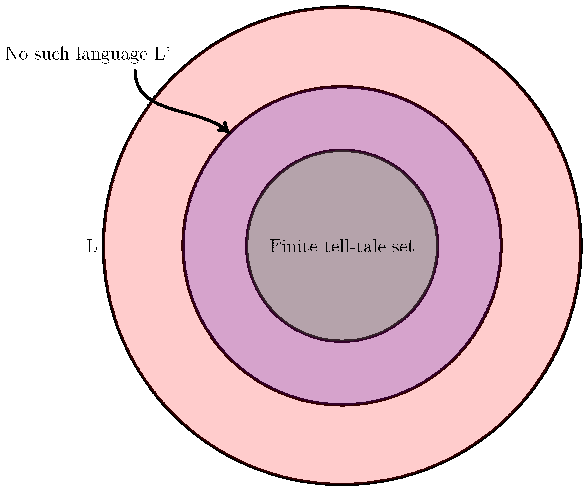


Figure 3.1: No such L' in every class identifable in the limit from positive data!

Corollary 1. *Every finite class of languages is identifiable in the limit from positve data.*

Gold's theorems and Angluin's theorems above are the basis for the so-called "Subset problem" in linguistics literature on learning (Wexler and Culicover, 1980; Berwick, 1985).

3.2 Identification in the limit from positive and negative data

A **positive and negative presentation** of a stringset S provides example strings not in S in addition to example strings in S. This can be formalized using the *characteristic function*

of S. Every set S has a characteristic function with domain Σ^* defined as follows.

$$f_S(s) = \begin{cases} 1 & \text{iff } s \in S \\ 0 & \text{otherwise} \end{cases}$$

Characteristic functions are total functions, which means defined for all $s \in \Sigma^*$. Also recall, that we write $(x, y) \in f$ whenever $f(x) = y$. So we can think of f_S as a set of points where $(s, 0)$ means $s \notin S$ and $(s, 1)$ means $s \in S$.

Then a **positive and negative presentation** of a stringset S is a function $\varphi : \mathbb{N} \to f_S$ such that φ is onto. Here, this means for every string $s \in \Sigma^*$, there is some $n \in \mathbb{N}$ such that $\varphi(n) = (s, f_s(s))$.

Definition 6 (Identification in the limit from positive and negative data)**.**

1 Algorithm A *identifies in the limit from positive and negative data* a class of stringsets
2 C provided
3 for all stringsets $S \in C$,
4 for all positive and negative presentations φ of S,
5 there is some number $n \in \mathbb{N}$ such that
6 for all $m > n$,
7 • the program output by A on $\varphi\langle m \rangle$ is the same as the the program
8 output by A on $\varphi\langle n \rangle$, and
9 • the program output by A on $\varphi\langle m \rangle$ solves the membership problem
10 for S.

The only difference between the definition above and the one in Definition 1 is in line 3. This paradigm is also called **learning from an informant.**

Theorem 8. *The computable class of languages is identifiable in the limit from positive and negative data.*

Proof[sketch] The algorithm proceeds by enumeration over programs. Since programs are strings, we can essentially use the enumeration for strings we used before.

The learning algorithm finds the first program in the enumeration that successfully classifies all of the data it has observed so far in φ.

How does it do this? Well, it looks at the first program in the enumeration and submits to it each data point $\varphi[i]$. If the first program fails to compute anything, or classifies any one of the data points incorrectly, the learning algorithm moves to the next program and checks again. This repeats. Evenutally, it must find a program which classifies all of the observed data points correctly. At this point, it outputs this program.

How do we know this algorithm converges to a correct program for S? Well, there is a program for S in the enumeration. There may be more than one such program so let P be the first program in the enumeration for S. Once the learning algorithm reaches P it will

output P since P will classify all data points from φ correctly because φ is a positive and negative presentation of S.

How do we know the learning algorithm will eventually output P on any positive and negative data presentation φ for S? Consider any program P' prior to P in the enumeration. Since P is the first program in the enumeration for S, P' is not a program for S. It follows that there is some $w \in S$, or $w \notin S$ that P' misclassifies. It follows that there is some point in time i such that $\varphi(i) = (w, x)$ with $x \in \{0, 1\}$. At this point the learning algorithm will conclude that P' does not classify everything it has seen in $\varphi[i]$ correctly, and will move to the next program in the enumeration. Since P' was arbitrary, it follows that the algorithm will eventually reach adn output program P. □

3.3 Identification in the limit from primitive recursive texts

Recall that algorithms have to succeed for *any* text or from *any* informant. As we discuss later, this has been one source of criticism of Gold's learning paradigm.

Let's consider texts for the moment. That is, let's consider positive presentations for some stringset S which has at least two strings in it. How many positive presentations are there? It should be easy to see that there are infinitely many. If u, v are distinct words in S then a text could start either u, v or u, u, v or u, u, u, v or u, u, u, u, v and so on. In fact, there are *uncountably many* presentations for S. This can be shown by the same diagnolization argument that we used earlier to show that there are uncountably many subsets of Σ^*.

We can also ask how many of these presentations are computable? Provided there are at least two strings in S, the answer is only countably many.

This situation is exactly analgous to the number of real numbers between 0 and 1, inclusive, and the number of computable real numbers between 0 and 1, inclusive.

In other words, *most* of the presentations that the Gold paradigm is required to succees on are *uncomputable*. Some have argued this is not reasonable.

Regardless of whether it is or not, we may be interested in what changes if we change the definition of learning to only require success on computable texts.

A particularly strong form of computability is computability via *primitive recursion*. This is weaker than Turing-machine computable. For example, Turing machines are not guaranteed to halt on input, but primitive recursive programs are guaranteed to halt. See Rogers (1967) for details on primitive recursion.

Definition 7 (Identification in the limit from primitive recursive texts)**.**

Algorithm A *identifies in the limit from positive data* a class of stringsets C provided
for all stringsets $S \in C$,
for all positive, computable presentations φ of S,
there is some number $n \in \mathbb{N}$ such that
for all $m > n$,
- the program output by A on $\varphi\langle m \rangle$ is the same as the the program output by A on $\varphi\langle n \rangle$, and
- the program output by A on $\varphi\langle m \rangle$ solves the membership problem for S.

The only difference between the definition above and the ones in Definitions 1 and 6 is in line 3.

Theorem 9. *The recursive (computable) class of languages is identifiable in the limit from primitive recursive texts.*

I'm not able at present to explain this proof, so it is omitted. I think the basic ideas are that (1) primitive recursive texts are enumerable and (2) it is possible to translate a primitive recursive text into a grammar a language equal to the content of the text (set of strings in the text). So an algorithm can identify by enumeration the primitive recursive text and thus the language the text is from.

3.4 Gold's interpretation of these results

From (Heinz, 2016):

Gold (1967:453-454) provides three ways to interpret his three main results:

1. The class of natural languages is much smaller than one would expect from our present models of syntax. That is, even if English is context-sensitive, it is not true that any context-sensitive language can occur naturally... In particular the results on [identification in the limit from positive data] imply the following: The class of possible natural languages, if it contains languages of infinite cardinality, cannot contain all languages of finite cardinality.
2. The child receives negative instances by being corrected in a way that we do not recognize...
3. There is an a priori restriction on the class of texts [presentations of data; i.e. infinite sequences of experience] which can occur...

The first possibility follows directly from the fact that no superfinite class of languages is identifiable in the limit from positive data. The second and third possibilities follow from Gold's other results on *identification in the limit from positive and negative data* and on *identification in the limit from positive primitive recursive data* ...

Each of these research directions can be fruitful, if honestly pursued. For the case of language acquisition, Gold's three suggestions can be investigated empirically. We ought to ask

1. What evidence exists that possible natural language patterns form subclasses of major regions of the Chomsky Hierarchy?
2. What evidence exists that children receive positive and negative evidence in some, perhaps implicit, form?
3. What evidence exists that each stream of experience each child is exposed to is guaranteed to be generated by a fixed, computable process (i.e. computable probability distribution or primitive recursion function)? More generally, what evidence exists that the data presentations are a priori limited?

My contention is that we have plenty of evidence with respect to question (1), some evidence with respect to (2), and virtually no evidence with respect to (3).

Finally, Gold concludes his paper this way.

> Concerning inductive inference, philosophers often occupy themselves with the following type of question: Suppose we are given a body of information and a set of possible conclusions, from which we are to choose one. Some of the conclusions are eliminated by the information. The question is, of the conclusions which are consistent with the in- formation, which is "correct"?
>
> If some sort of probability distribution is imposed on the set of conclusions, then the problem is meaningful. But if no basis for choosing between the consistent conclusions is postulated a priori, then inductive inference can do no more than state the set of consistent conclusions.
>
> The difficulty with the inductive inference problem, when it is stated this way, is that it asks, "What is the correct guess at a specific time with a fixed amount of information?" There is no basis for choosing between possible guesses at a specific time. However, it is interesting to study a guessing strategy. Now one can investigate the limiting behavior of the guesses as successively larger bodies of information are considered. This report is an example of such a study. Namely, in interesting identification problems, a learner cannot help but make errors due to incomplete knowledge. But, using an "identification in the limit" guessing rule, a learner can guarantee that he will be wrong only a finite number of times.

3.5 Criticisms

1. Identification in the limit from positive data is too hard. The texts can be adversarial.
2. identification in the limit doesn't address time or resource complexity of learning.

The first point is articulated well by Clark and Lappin (2011).

The second point is about feasibility. Learning by enumeration is very, very far from efficient. So even if every finite class of languages is identifiable in the limit from positive

data, large finite classes may not be efficiently learnable because learning by enumeration is awfully slow! Similarly, Even if the recursive class is identifiable in the limit from primitive recursive text, it is not efficiently learnable. So we need some way to identify feasibly learnable subclasses.

Much research since Gold has aimed to incorporate feasibility into learning. The Probably Approximately Correct learning model is one influential example (Valiant, 1984; Anthony and Biggs, 1992; Kearns and Vazirani, 1994).

Many researchers advocate a learning setting where the aim is not to learn categorical stringsets but to learn probabilty distributions over them ("stochastic stringsets.") We will talk about this next.

The most repeated refrain ever in cognitive scince, computational linguistics about the theory of learning languages is this: "Gold (1967) showed that context-free grammars are not learnable but Horning (1969) showed that probabilistic context-free grammars are." There is so much confusion about this, I wrote about it: (Heinz, 2016).

Chapter 4

Automata Methods

In this chapter we explain how using finite-state automata as grammar formalism, can lead to successful learning algorithms. A key aspect of the results here is that the automata are deterministic.

In the first section, we examine cases where the structure of the automata is fixed. We review Strictly k-Local learaning in the context of finite-state automata. We see how the method there can be generalized to stochastic stringsets and transductions. The stochastic Strictly k-Local stringsets are n-gram models in the NLP literature. Input and Output Strictly Local transductions are due to Chandlee (Chandlee, 2014; Chandlee *et al.*, 2014, 2015; Chandlee and Heinz, Forthcoming; Chandlee *et al.*, To appear).

In the second section, we examine cases where the structure of the automata is not fixed. The primary result there is the alogorithm RPNI which can learn any regular language from positve and negative data in polynomial time and data. This result provides the basis for algorithms which efficiently learn any stochastic regular stringets (ALEGRIA) and any sequential transductions (OSTIA) from positive data only. The reason why ALEGRIA and OSTIA can get by with positive data but RPNI cannot is discussed.

4.1 Finite-state automata

Definition 8. A *deterministic finite-state acceptor (DFA)* is a tuple $(Q, \Sigma, q_0, F, \delta)$ where

- Q is a finite set of states;
- Σ is a finite set of symbols (*the alphabet*);
- $q_0 \in Q$ is the *initial* state;
- $F \subseteq Q$ is a set of *accepting* (*final*) states; and
- δ is a function with domain $Q \times \Sigma$ and co-domain Q. It is called the *transition function*.

We extend the domain of the transition function to $Q \times \Sigma^*$ as follows. In these notes, the empty string is denoted with λ.

$$\begin{aligned} \delta^*(q, \lambda) &= q \\ \delta^*(q, aw) &= \delta^*((\delta(q, a), w) \end{aligned} \tag{4.1}$$

Consider some DFA $A = (Q, \Sigma, q_0, F, \delta)$ and string $w \in \Sigma^*$. If $\delta^*(q_0, w) \in F$ then we say A *accepts/recognizes/generates* w. Otherwise A *rejects* w.

Definition 9. The stringset recognized by A is $L(A) = \{w \in \Sigma^* \mid \delta^*(q_0, w) \in F\}$.

The use of the 'L' denotes "Language" as stringsets are traditionally referred to as *formal languages*.

Definition 10. A stringset is *regular* if there is a DFA that recognizes it.

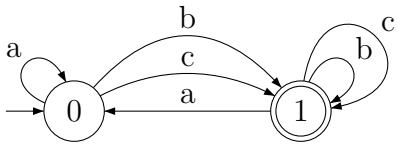
4.1.1 Exercises

Exercise 14. This exercise is about designing DFA. Let $\Sigma = \{a, b, c\}$. Write DFA which express the following generalizations on word well-formedness.

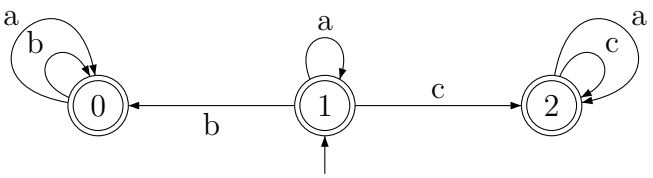
1. All words begin with a consonant, end with a vowel, and alternate consonants and vowels.
2. Words do not contain *aaa* as a substring.
3. If a word begins with a, it must end with c.
4. Words must contain two bs.
5. Words have an even number of nasals (Let $\Sigma = \{a, b, n\}$).

Exercise 15. This exercise is about reading and interpreting DFA. Provide generalizations in English prose which accurately describe the stringset these DFA describe.

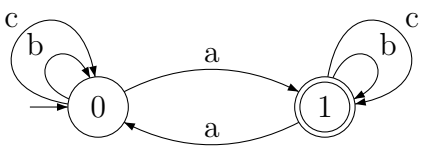
1.



2.



3.



4. Write the DFA in #1-3 in mathematical notation. So what is Q, Σ, q_0, F, and δ?

4.2 Generalizing Strictly Locality with Weighted Automata

For each k, and each there is a canonical form for strictly k-local stringsets. Here it is.

Definition 11. Given the stucture of the *canonical SL_k DFA* for $L(G)$ is a DFA such that
- $Q = \Sigma^{\leq k-1}$;
- Σ is the alphabet;
- $q_0 = \lambda$;
- $F = Q$; and
- For all $q \in Q, \sigma \in \Sigma$, $\delta(q, \sigma) = \texttt{suff}_{k-1}(qa)$

Here is an example, where $\Sigma = \{a, b, c\}$ and $k = 2$. Observe that the language of this DFA is Σ^*.

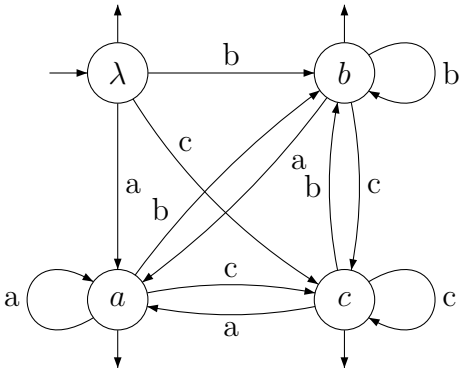


Figure 4.1: A canonical SL_2 DFA

Other SL_2 languages are obtained by removing transitions or making states non-final. In other words, every SL_2 stringset corresponds to some subgraph of this canonical DFA.

To see why, consider any strictly k-local grammar $G \subseteq \texttt{factor}_k(\{\rtimes\}\Sigma^*\{\ltimes\})$.
- For all $w \in \Sigma^*$: $w \in F$ iff $w\ltimes \in G$ or $\rtimes w\ltimes \in G$.
- For all $wa \in \Sigma^*$: $\delta(w, a)$ exists iff $wa \in G$ or $\rtimes wa \in G$.

4.2.1 Strictly k-Local stringsets

We have already seen an algorithm that learns Strictly k-local stringsets. It operates by recording the k-factors of the observed words appended with word boundaries. These k-factors correspond to transitions in the canonical machine. So as each word is observed, we essenentially carve a path in the canonical SL_k DFA.

Figure 4.2 illustrates such learning for positive presentations beginning with $\langle a, aaab, bc\rangle$.

It is worthwhile to compare the development of the DFA above compared to string extension learning grammar.

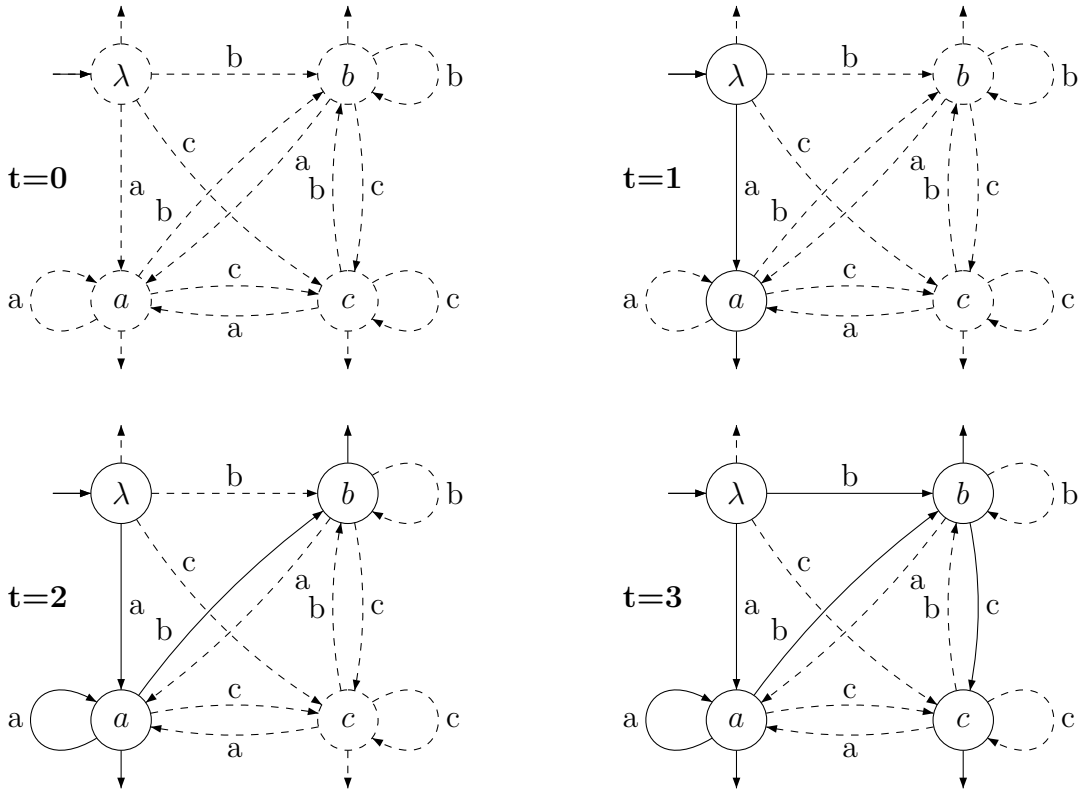


Figure 4.2: Learning as carving paths in the pre-existing structure. The DFA at time $t = 0$ shows is the structure and subsequent DFA show the paths carved at each point in time with $\varphi(1) = a$, $\varphi(2) = aaab$, $\varphi(2) = bc$.

time t	$\varphi(t)$	$\texttt{factor}_k(\rtimes\varphi(t)\ltimes)$	G_t
0			$\emptyset$
1	a	{⋊a, a⋉}	{⋊a, a⋉}
2	aaab	{⋊a, aa, ab, b⋉}	{⋊a, a⋉, aa, ab, b⋉}
3	bc	{⋊b, bc, c⋉}	{⋊a, a⋉, aa, ab, b⋉, ⋊b, bc, c⋉}

For each time t, the stringset generated by the grammar and the DFA are exactly the same!

4.2.2 Stochastic Strictly k-Local stringsets

"Carving paths" in a fixed determinstic structure can be generalized to learn stochastic stringsets. A stochastic stringset is a probability distribution over Σ^*. So each string has some probability between 0 and 1 (inclusive) and the sum of all the probabilities of all the strings equals 1.

$$\sum_{w \in \Sigma^*} P(w) = 1 \tag{4.2}$$

Each DFA describes a class of stochastic stringsets as follows. Each transition $\delta(q, a)$ is associated with a probability θ_{qa}. Each state is also associated with a probability $\theta_{q\ltimes}$. The probability on the state is the probability of stopping/accepting/finality. The probabilities associated with each transitiona and each state are the *parameters* of the model the DFA describes. Provided for each state q, the following holds then the DFA describes a probability distribution over Σ^*.

$$\sum_{a \in \Sigma} \theta_{qa} + \theta_{q\ltimes} = 1 \tag{4.3}$$

We define a function π which describes the computational "process" and "path" string $w \in \Sigma^*$ takes from any state $q \in Q$ with any initial real value r. The operator $(\cdot)$ indicates multiplication.

$$\begin{aligned} \pi(q, \lambda, r) &= r \cdot \theta_{q\ltimes} \\ \pi(q, wa, r) &= \pi(\delta(q, a), w, r \cdot \theta_{qa}) \end{aligned} \tag{4.4}$$

The function such a real-weighted DFA A describes is given by the equation below.

$$f_A(w) = \pi(q_0, w, 1) \tag{4.5}$$

It can be said that the DFA is a *parametric* model with $|Q| + |Q| \cdot |\Sigma|$ parameters. For a given DFA whose parameters are fixed, these parameters are often written simply as θ even if there are many of them. You can think of θ as a vector of values. The possible values these parameters can take on while still ensuring a probability distribution over Σ^* is denoted Θ.

So what does it mean to learn a stochastic stringset? Given a parametric model, it basically means finding the true values of the parameters θ from the set of all possible parameter settings Θ. This may not reasonable when the data sample is small. So here is one definition of learning stochastic stringsets that has been proposed that avoids the issue of data sufficiency.

Definition 12 (Maximum Likelihood Estimate (MLE))**.**

Algorithm A *yields the maximum likelihood estimate* for a class of stochastic stringsets C provided

for all stochastics stringsets $S \in C$,
for all sequences D of independent and identically distributed strings drawn from S
- the parameters θ output by A assign a probability to D which is greater than the probability other parameter choices $\theta' \in \Theta$ assign to D.

In other words, if A outputs parameters θ than any deviation from θ will lower the probability of the data. In this way, the learning algorithm does not try to find the true parameter values,

it simply identifies those values that maximize the likelihood (probability) of the data under the assumed parametric model.

Importantly, the MLE is also what is known as a *consistent* estimator. With enough data, it *will* converge to the true values. Calculus expresses convergence over real values with the idea of aribtirary precision. So for any small number ϵ you think of, there will be some large sample of data such that the MLE will return parameters θ that assign a probability to words that are within ϵ of the true values.

Definition 13 (Consistent Estimator)**.**

Algorithm A is a *consistent estimator* for a class of stochastic stringsets C provided

for all stochastics stringsets $S \in C$,
for all $\epsilon > 0$
there is some number $n \in \mathbb{N}$ such that
for all sequences D of independent and identically distributed strings drawn from S with $|D| > n$,
- the parameters θ output by A with D are within ϵ of the true parameter values.

OK, that's a measuring stick by which we can judge our learning algorithms.

If the parametric is a DFA, as we have considered above, then the algorithm that

1. pushes the data through the DFA,
2. counts the times each transition is traversed, and then
3. then normalizes the counts to obtain parameter values

yields the maximum likelihood estimate. This result is a proven theorem (Vidal *et al.*, 2005a,b).

Figure 4.3 shows how the counting would progress and Table 4.1 how the parameter values are updated over time.

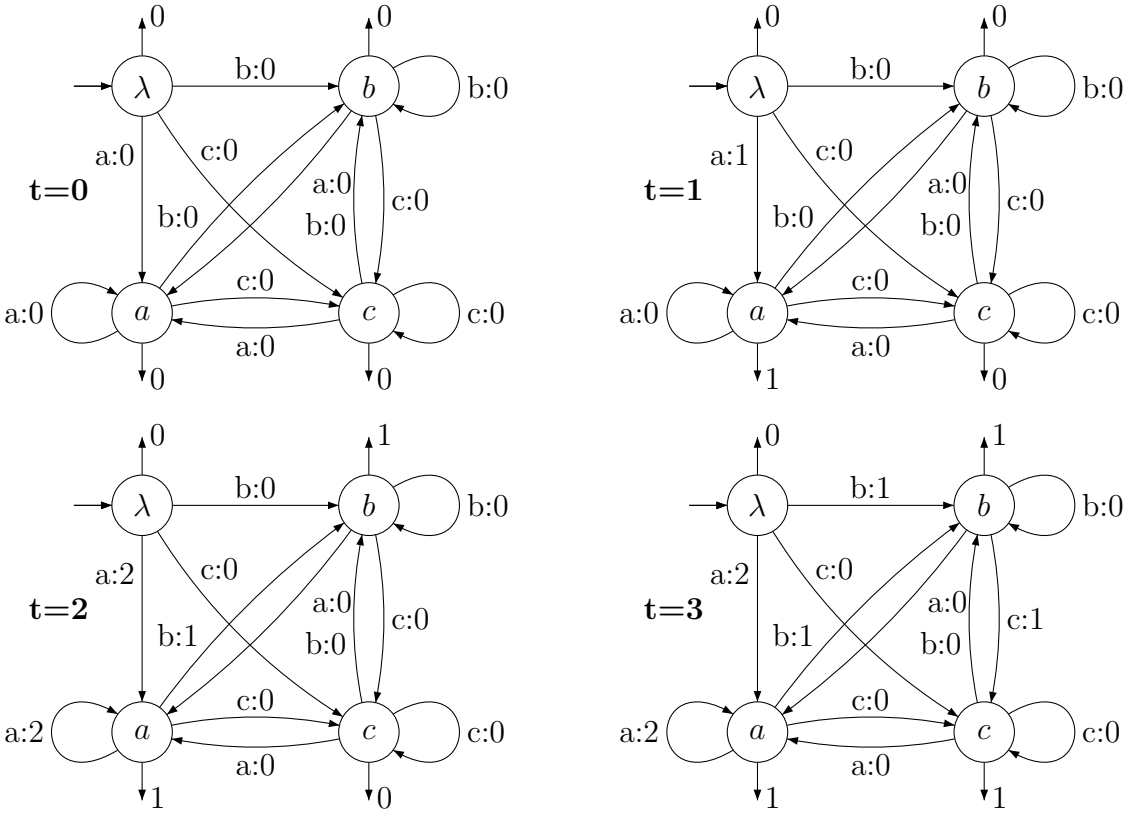


Figure 4.3: Learning as *counting* paths in the pre-existing structure. The DFA at time $t = 0$ shows is the structure and subsequent DFA show the paths carved at each point in time with $\varphi(1) = a$, $\varphi(2) = aaab$, $\varphi(2) = bc$.

	0	1	2	3		0	1	2	3
$\theta_{\lambda\ltimes}$	0	0	0	0	$\theta_{b\ltimes}$	0	0	1	0.5
$\theta_{\lambda a}$	0	1	1	0.67	θ_{ba}	0	0	0	0
$\theta_{\lambda b}$	0	0	0	0.33	θ_{bb}	0	0	0	0
$\theta_{\lambda c}$	0	0	0	0	θ_{bc}	0	0	0	0.5
$\theta_{a\ltimes}$	0	1	0.25	0.25	$\theta_{c\ltimes}$	0	0	0	1
θ_{aa}	0	0	0.5	0.5	θ_{ca}	0	0	0	0
θ_{ab}	0	0	0.25	0.25	θ_{cb}	0	0	0	0
θ_{ac}	0	0	0	0	θ_{cc}	0	0	0	0

Table 4.1: Parameter values for the SL_2 parametric model with the sample

4.2.3 Input Strictly k-Local Transductions

Tracing the paths through a given deterministic finite-state automaton also helps us understand how transductions can be learned. A string-to-string transduction is a function from Σ^* to Δ^*.

Each DFA describes a class of transductions as follows. Each transition $\delta(q, a)$ is associated with an output string θ_{qa}. Each state is also associated with an output string $\theta_{q\ltimes}$, and the initial state is also associated with an output string $\theta_{\rtimes}$. The strings associated with each transitions and each state are the *parameters* of the model the DFA describes.

We define a function π which describes the computational "process" and "path" string $w \in \Sigma^*$ takes from any state $q \in Q$ with any initial string value $v \in \Delta^*$. The operator $(\cdot)$ refers to concatenation.

$$\begin{aligned} \pi(q, \lambda, v) &= v \cdot \theta_{q\ltimes} \\ \pi(q, wa, v) &= \pi(\delta(q, a), w, v \cdot \theta_{qa}) \end{aligned} \tag{4.6}$$

Then the function such a string-weighted DFA A describes is given by the equation below.

$$f_A(w) = \pi(q_0, w, \theta_{\rtimes}) \tag{4.7}$$

Figure 4.4 shows two Input Strictly 2-Local functions with $\Sigma = \Delta = \{a, b, c\}$. The first function describes "progressive b-assimilation." As a rewrite rule, this process would be expressed as $c \rightarrow b / b\_\_$. The second function describes "regressive b-assimilation." As a rewrite rule, this process would be expressed as $c \rightarrow b / \_\_ b$.

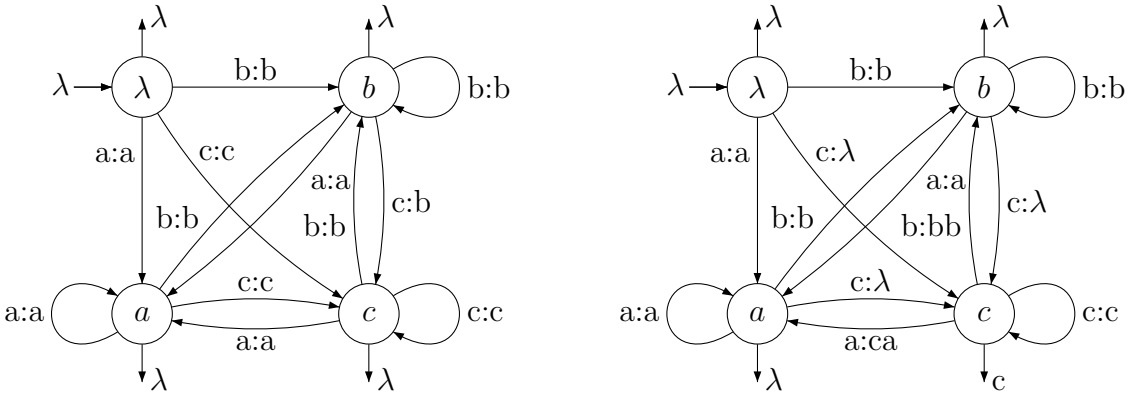


Figure 4.4: 2-ISL transducers for progressive b-assimilation (left) and regressive b-assimilation (right).

The identification in the limit paradigm for positive data natrually yields a definitions transduction learning. A positive presentation is now example transductions, which are input-output pairs $(w, f(w))$.

More precisely, a **positive presentation** of a string-to-string function f is a function $\varphi : \mathbb{N} \to f$ such that φ is onto. This means for every input-output pair $(w, f(w))$ defined by f, there is some $n \in \mathbb{N}$ such that $\varphi(n) = (w, f(w))$.

Definition 14 (Identification in the limit from positive data (function version))**.**

Algorithm A *identifies in the limit from positive data* a class of string-to-string functions C provided

for all functions $f \in C$,
for all positive presentations φ of f,
there is some number $n \in \mathbb{N}$ such that
for all $m > n$,
- the program output by A on $\varphi\langle m \rangle$ is the same as the the program output by A on $\varphi\langle n \rangle$, and
- the program output by A on $\varphi\langle m \rangle$ which takes any string $w \in \Sigma^*$ for which f is defined as input and returns $f(w)$ as output.

The algorithm SOSFIA (Structured Onward Subsequential Function Inference Algorithm) (Jardine *et al.*, 2014) provably identifies the k-ISL functions in the limit from positive data.

Here is a summary of how SOSFIA works with illustrations by examples. Given a sample S of input-output pairs, SOSFIA calculates the *common output* (`common_out`) of every prefix of any input string. The common output of an input prefix u is the longest prefix common to all the output strings whose corresponding input strings have prefix u. This *longest common prefix* is denoted `lcp`.

SOSFIA then uses these common outputs to calculate the *minimal change* (`min_change`) each letter introduces to the output string. These minimal changes are the parameter values (the outputs associated with the transitions). Minimal change is calculated using "left division." This operation "strips away" a prefix of a string. Formally, whenever $w = uv$ then $u^{-1}w = v$. We say "the left division of w by u equals v."

Formal definitions of `common_out` and `min_change` from Jardine *et al.* (2014, p. 101).

Definition 15. The *common output* of an input prefix w in a sample $S \subset \Sigma^* \times \Delta^*$ for t is the `lcp` of all $t(wv)$ that are in S: $\texttt{common\_out}_S(w) = \texttt{lcp}(\{u \in \Sigma^* \mid \exists v \text{ s.t. } (wv, u) \in S\})$

Definition 16. The *minimal change in the output* in $S \subset \Sigma^* \times \Delta^*$ from w to $w\sigma$ is:

$$\texttt{min\_change}_S(\sigma, w) = \begin{cases} \texttt{common\_out}_S(\sigma) & \text{if } w = \lambda \\ \texttt{common\_out}_S(w)^{-1}\texttt{common\_out}_S(w\sigma) & \text{otherwise} \end{cases}$$

Consider progressive b-assimilation and let the sample S be as shown.

$$S = \{(aa, aa), (ab, ab), (ac, ac), (ba, ba), (bb, bb), (bc, bb), (ca, ca), (cb, cb), (cc, cc)\}$$

The input prefixes in this sample are $S_i = \{\lambda, a, b, c, aa, ab, ac, ba, bb, bc, ca, cb, cc\}$. So we need to calculate the longest common prefix of all the outputs associated with each string. Here they are.

$\texttt{common\_out}_S(\lambda)$	$=$	$\texttt{lcp}(f(S_i))$	$=$	λ
$\texttt{common\_out}_S(a)$	$=$	$\texttt{lcp}(f(a), f(aa), f(ab), f(ac))$	$=$	a
$\texttt{common\_out}_S(b)$	$=$	$\texttt{lcp}(f(b), f(ba), f(bb), f(bc))$	$=$	b
$\texttt{common\_out}_S(c)$	$=$	$\texttt{lcp}(f(c), f(ca), f(cb), f(cc))$	$=$	c
$\texttt{common\_out}_S(aa)$	$=$	$\texttt{lcp}(f(aa))$	$=$	aa
$\texttt{common\_out}_S(ab)$	$=$	$\texttt{lcp}(f(ab))$	$=$	ab
$\texttt{common\_out}_S(ac)$	$=$	$\texttt{lcp}(f(ac))$	$=$	ac
$\texttt{common\_out}_S(ba)$	$=$	$\texttt{lcp}(f(ba))$	$=$	ba
$\texttt{common\_out}_S(bb)$	$=$	$\texttt{lcp}(f(bb))$	$=$	bb
$\texttt{common\_out}_S(bc)$	$=$	$\texttt{lcp}(f(bc))$	$=$	bb
$\texttt{common\_out}_S(ca)$	$=$	$\texttt{lcp}(f(ca))$	$=$	ca
$\texttt{common\_out}_S(cb)$	$=$	$\texttt{lcp}(f(cb))$	$=$	cb
$\texttt{common\_out}_S(cc)$	$=$	$\texttt{lcp}(f(cc))$	$=$	cc

With the `common_out` values we can calculate the minimal changes.

$\texttt{min\_change}_S(a, \lambda)$	$=$	$\texttt{common\_out}_S(a)$			$=$	a	
$\texttt{min\_change}_S(b, \lambda)$	$=$	$\texttt{common\_out}_S(b)$			$=$	b	
$\texttt{min\_change}_S(c, \lambda)$	$=$	$\texttt{common\_out}_S(c)$			$=$	c	
$\texttt{min\_change}_S(a, a)$	$=$	$\texttt{common\_out}_S(a)^{-1}\texttt{common\_out}_S(aa)$	$=$	$a^{-1}aa$	$=$	a	
$\texttt{min\_change}_S(b, a)$	$=$	$\texttt{common\_out}_S(a)^{-1}\texttt{common\_out}_S(ab)$	$=$	$a^{-1}ab$	$=$	b	
$\texttt{min\_change}_S(c, a)$	$=$	$\texttt{common\_out}_S(a)^{-1}\texttt{common\_out}_S(ac)$	$=$	$a^{-1}ac$	$=$	c	
$\texttt{min\_change}_S(a, b)$	$=$	$\texttt{common\_out}_S(b)^{-1}\texttt{common\_out}_S(ba)$	$=$	$b^{-1}ba$	$=$	a	
$\texttt{min\_change}_S(b, b)$	$=$	$\texttt{common\_out}_S(b)^{-1}\texttt{common\_out}_S(bb)$	$=$	$b^{-1}bb$	$=$	b	
$\texttt{min\_change}_S(c, b)$	$=$	$\texttt{common\_out}_S(b)^{-1}\texttt{common\_out}_S(bc)$	$=$	$b^{-1}bb$	$=$	b	(!!)
$\texttt{min\_change}_S(a, c)$	$=$	$\texttt{common\_out}_S(c)^{-1}\texttt{common\_out}_S(ca)$	$=$	$c^{-1}ca$	$=$	a	
$\texttt{min\_change}_S(b, c)$	$=$	$\texttt{common\_out}_S(c)^{-1}\texttt{common\_out}_S(cb)$	$=$	$c^{-1}cb$	$=$	b	
$\texttt{min\_change}_S(c, c)$	$=$	$\texttt{common\_out}_S(c)^{-1}\texttt{common\_out}_S(cc)$	$=$	$c^{-1}cc$	$=$	c	

The minimal change letter σ with string w gives us the output string at the state $\texttt{suff}_{k-1}(w)$ for the transition labeled σ. Above, we would have $\texttt{min\_change}_S(\sigma, q) = \theta_{q\sigma}$.

Now consider regressive b-assimilation and let the sample S be as shown.

$$S = \left\{ \begin{array}{l} (aa, aa), (ab, ab), (ac, ac), (ba, ba), (bb, bb), (bc, bc), (ca, ca), (cb, bb), (cc, cc), \\ (aca, aca), (acb, acb), (bca, bca), (bcb, bbb), (cca, cca), (ccb, cbb) \end{array} \right\}$$

As before, the input prefixes in this sample are $S_i = \{\lambda, a, b, c, aa, ab, ac, ba, bb, bc, ca, cb, cc\}$. So we need to calculate the longest common prefix of all the outputs associated with each string. Here they are.

$\texttt{common\_out}_S(\lambda)$	$=$	$\texttt{lcp}(f(S_i))$	$=$	λ	
$\texttt{common\_out}_S(a)$	$=$	$\texttt{lcp}(f(a), f(aa), f(ab), \ldots)$	$=$	a	
$\texttt{common\_out}_S(b)$	$=$	$\texttt{lcp}(f(b), f(ba), f(bb), \ldots)$	$=$	b	
$\texttt{common\_out}_S(c)$	$=$	$\texttt{lcp}(f(c), f(ca), f(cb), \ldots)$	$=$	λ	(!!)
$\texttt{common\_out}_S(aa)$	$=$	$\texttt{lcp}(f(aa))$	$=$	aa	
$\texttt{common\_out}_S(ab)$	$=$	$\texttt{lcp}(f(ab))$	$=$	ab	
$\texttt{common\_out}_S(ac)$	$=$	$\texttt{lcp}(f(ac), f(aca), f(acb))$	$=$	a	(!!)
$\texttt{common\_out}_S(ba)$	$=$	$\texttt{lcp}(f(ba))$	$=$	ba	
$\texttt{common\_out}_S(bb)$	$=$	$\texttt{lcp}(f(bb))$	$=$	bb	
$\texttt{common\_out}_S(bc)$	$=$	$\texttt{lcp}(f(bc), f(bca), f(bcb))$	$=$	b	(!!)
$\texttt{common\_out}_S(ca)$	$=$	$\texttt{lcp}(f(ca))$	$=$	ca	
$\texttt{common\_out}_S(cb)$	$=$	$\texttt{lcp}(f(cb))$	$=$	bb	
$\texttt{common\_out}_S(cc)$	$=$	$\texttt{lcp}(f(cc), f(cca), f(ccb))$	$=$	c	(!!)

With the `common_out` values we can calculate the minimal changes.

$\texttt{min\_change}_S(a, \lambda)$	$=$	$\texttt{common\_out}_S(a)$			$=$	a
$\texttt{min\_change}_S(b, \lambda)$	$=$	$\texttt{common\_out}_S(b)$			$=$	b
$\texttt{min\_change}_S(c, \lambda)$	$=$	$\texttt{common\_out}_S(c)$			$=$	c
$\texttt{min\_change}_S(a, a)$	$=$	$\texttt{common\_out}_S(a)^{-1}\texttt{common\_out}_S(aa)$	$=$	$a^{-1}aa$	$=$	a
$\texttt{min\_change}_S(b, a)$	$=$	$\texttt{common\_out}_S(a)^{-1}\texttt{common\_out}_S(ab)$	$=$	$a^{-1}ab$	$=$	b
$\texttt{min\_change}_S(c, a)$	$=$	$\texttt{common\_out}_S(a)^{-1}\texttt{common\_out}_S(ac)$	$=$	$a^{-1}a$	$=$	λ
$\texttt{min\_change}_S(a, b)$	$=$	$\texttt{common\_out}_S(b)^{-1}\texttt{common\_out}_S(ba)$	$=$	$b^{-1}ba$	$=$	a
$\texttt{min\_change}_S(b, b)$	$=$	$\texttt{common\_out}_S(b)^{-1}\texttt{common\_out}_S(bb)$	$=$	$b^{-1}bb$	$=$	b
$\texttt{min\_change}_S(c, b)$	$=$	$\texttt{common\_out}_S(b)^{-1}\texttt{common\_out}_S(bc)$	$=$	$b^{-1}b$	$=$	λ
$\texttt{min\_change}_S(a, c)$	$=$	$\texttt{common\_out}_S(c)^{-1}\texttt{common\_out}_S(ca)$	$=$	$\lambda^{-1}ca$	$=$	ca
$\texttt{min\_change}_S(b, c)$	$=$	$\texttt{common\_out}_S(c)^{-1}\texttt{common\_out}_S(cb)$	$=$	$\lambda^{-1}bb$	$=$	bb
$\texttt{min\_change}_S(c, c)$	$=$	$\texttt{common\_out}_S(c)^{-1}\texttt{common\_out}_S(cc)$	$=$	$\lambda^{-1}c$	$=$	c

Again, we see that $\texttt{min\_change}_S(\sigma, q) = \theta_{q\sigma}$.

Readers are referred to the paper for full details on SOSFIA and that it provably identifies in the limit the class of k-ISL functions.

4.2.4 Output Strictly k-Local Transductions

Chandlee *et al.* (2015) prove a similar algorithm for inferring k-OSL functions.

4.3 Generalizing to any DFA

The aforementioned strategies hold for *any* deterministic finite-state automata. In other words, each deterministic finite state machine defines a class of stringsets, a class of stochastic stringsets, and a class of transductions, and each class can be learned with the methods described above.

Heinz and Rogers (2013) establish this for Boolean case. For stochastic stringsets, this result was known much earlier. Jardine *et al.* (2014) establish this for "Input" based transductions. For output-based transductions, the theorems Chandlee *et al.* (2015) do not address this general case, but the same techniques apply there as well.

To my knowledge, these results have not yet been ported to tree automata.

Chapter 5

Summary of Part 1

Let us review what we have covered so far.

5.1 Computational characterizations of linguistic generalizations

1. It is important to be able to relate the intensional description of a linguistic generalization (a grammar) to its extensional description (a potentially infinite set of points).
2. This requires some mathematics.
3. Formal grammars like automata and logic are well-studied tools that accomplish this.
4. They provide insights into the nature of natural language patterns not obtainable in other ways.
5. They are not finished! There is a lot yet to accomplish to develop formal grammars for linguistics, and we should not ignore the lessons of previous research.
6. For linguistics:
   - Well-formedness can be characterized with *sets*
   - Transformations can be characterized with *functions*
7. So what is the nature of these sets and functions for linguistics?

5.2 Algorithms

1. Algorithms are procedures which solve well-defined problems after finitely many steps.
2. Proving an algorithm solves a well-defined problem is important.
3. Proving an algorithm finds the answer to any instance of the problem with a certain amount of resources is also important.
4. This also requires some mathematics.
5. These results guarantee the *general* behavior of the algorithm.
6. This stands in contrast to simulations, which only show a program's specific behavior on a specific instance of some problem.

7. Problems for linguistics where algorithms help:
   - Membership problems
   - Transformation problems
   - Learning problems.

5.3 Defining Learning

1. What is a reasonable definition? Many issues and answers.
   - When is the data is good enough?
   - What counts as successful learning?
2. Learning problems ought to be defined for classes of generalizations, not individual generalizations.

5.4 Learning Definitions

1. Identification in the limit
   (a) Does the learner only make finitely many mistakes for any of the allowable presentations of examples?
   - from positive data on arbitrary presentations
   - from positive and negative data on arbitrary presentations
   - from positive data on recursive presentations
   - from positive data on primitive recursive presentations
2. Maximum likelihood estimate for stochastic sets
   (a) Does the learner do its best? (This means any deviation from its output worse.)
   - It is constrained by the data it gets.
   - It is constrained by its hypothesis space/structural limitations/parametric model/space of parametric models.
3. Other definitions we did get to.
   (a) Probably Approximately Correct learning
   (b) Maximizing the margin (for classification)
   (c) Maximizing the entropy (related to MLE)
   (d) Stochastic finite learning
   (e) ...
4. Complexity issues we did not get to.
   (a) Mistake bounds
   (b) VC dimension
   (c) Update-time and "Pitt's trick"
   (d) ...

5.5 Important results in learning theory

1. Finite class of stringsets is identifiable in the limit from positive data on arbitrary presentations.
2. No superfinite class of stringsets is identifiable in the limit from positive data on arbitrary presentations.
3. The computably enumerable stringsets are identifiable in the limit from positive and negative data on arbitrary presentations. (This learner by enumeration is not efficient.)
4. The computable stringsets are identifiable in the limit from positive data on primitive recursive presentations. (This learner by enumeration is not efficient.)
5. Gold's conclusions and critical analysis/reflection.
   - Maybe not all context-free (context-sensitive) stringsets are possible human languages.
   - Maybe humans access negative evidence in some way.
   - Maybe the data humans receive is not arbitrary in some way.
6. Results we did not go over:
   (a) The regular class of stringsets is efficiently identifiable in the limit from positive and negative data (RPNI).
   (b) Deterministic regular transductions are identifiable in the limit from positive data (OSTIA).
   (c) Deterministic regular probability distributions are identifiable in the limit from positive data (RLIPS, ALEGRIA).
   (d) Corresponding results extend these to tree languages and tree transductions.

5.6 String extension learning and automata learning

1. String extension learning
   (a) Basic idea: Well-formedness of a structure is determined by its "parts"
   (b) Formal grammars are finite sets.
      - substrings (SL)
      - subsequences (SP)
      - substrings on a tier (TSL)
      - sets of substrings (LT)
      - sets of subsequences (PT)
      - multisets of substrings (LTT)
      - local trees (SL treesets)
      - lots of possibilities ...
   (c) There is a function which maps structures like strings or trees to sets.
   (d) The formal grammar defines a set of structures (like strings or trees) as all structures whose image under the function is a subset of the grammar.
   (e) Learning builds a grammar by unioning the images of the examples under the function. (It begins with the empty set.)

   (f) Identification in the limit from positive data.
2. Automata learning stringsets
   (a) Basic idea: Memory required to determine well-formedness is independent of length of the input.
   (b) A finite-state machine is a grammar solving some membership or transformation problem.
   (c) Deterministic finite acceptors (DFAs) correspond to classes of stringsets.
      - Each SL_k stringset is a sub-graph of the SL_k DFA.
      - Each $\mathrm{TSL}_{T,k}$ stringset is a sub-graph of the $\mathrm{TSL}_{T,k}$ DFA.
      - Each LT_k stringset is a sub-graph of the LT_k DFA.
      - Each PT_k stringset is a sub-graph of the PT_k DFA.
      - Each $\mathrm{LTT}_{t,k}$ stringset is a sub-graph of the $\mathrm{LTT}_{t,k}$ DFA.
   (d) Learning simply traces the path of the sample in the DFA.
3. Automata learning stochastic stringsets
   (a) Probabilities are added to the transitions.
   (b) DFAs correspond to classes of stochastic stringsets as before.
   (c) Learning simply counts the paths of the sample in the DFA and normalizes.
   (d) Provably gets the MLE.
4. Automata learning string-to-string functions
   (a) Output strings are added to the transitions.
   (b) DFAs correspond to classes of string-to-string transformations as before.
   (c) Learning assigns to each transition the contribution (minimal change) each symbol makes at state q by factoring out the common output of all input strings which lead to q.
   (d) Provably efficiently learns the class of transductions.
5. Work in progress
   (a) SP_k is characterized by a *set* of DFAs and learning traces the sample through each DFA. (The stringset the set of DFAs defines is given by intersection.)
   (b) Heinz and Rogers (2010) generalized this idea to learn the MLE of stochastic SP_k stringsets.
   (c) Shibata and Heinz (in prep) provide a much better proof of this result, and generalize (I think) to sets of DFAs under some conditions.
   (d) The transducer case is wide open, though I have some ideas.

5.7 Open Questions

1. ISL and OSL functions are *functions*—so each input has at most one output.
   (a) How can we add variation to this?
   (b) How can we add probabilities to this?
   (c) Once accomplished, this has MANY potential applications in NLP.
2. The proper notion of k-factor and generalizing these results to representations
   (a) Subsequences are k-factors with the right representations

   (b) SL tree languages are k-factors with the right representations
   (c) What about k-factors of autosegmental structures?
   (d) What about k-factors for feature-based representations?
3. The subregular classes for strings and string-to-string functions can be lifted to trees.
   (a) SP, LT, PT, TSL treesets? What are these and do they capture syntactic generalizations?
   (b) ISL and OSL tree transductions?
4. Expanding the grammatical architecture
   (a) We have examined learning generalizations within individual modules of grammar (phonotactics, phonological transformations, morphological transformations, syntactic well-formedness).
   (b) How can more than one component be learned simultaneously?
   (c) What is the role or character of the lexicon in morpho-phonological learning?
   (d) How can learning be defined in the face of exceptions? (cf. Tolerance Principle)
   (e) What algorithms can "successfully learn" in the face of exceptions?
5. For all these questions, it is imperative to define a learning problem. What are the instances of the problem (example data)? What are its answers (target grammars)?

Bibliography

Angluin, Dana. 1980. Inductive inference of formal languages from positive data. *Information Control* 45:117–135.

Anthony, M., and N. Biggs. 1992. *Computational Learning Theory*. Cambridge University Press.

Berwick, Robert. 1985. *The acquisition of syntactic knowledge*. Cambridge, MA: MIT Press.

Chandlee, Jane. 2014. Strictly local phonological processes. Doctoral dissertation, The University of Delaware.

Chandlee, Jane, Rémi Eyraud, and Jeffrey Heinz. 2014. Learning strictly local subsequential functions. *Transactions of the Association for Computational Linguistics* 2:491–503.

Chandlee, Jane, Rémi Eyraud, and Jeffrey Heinz. 2015. Output strictly local functions. In *Proceedings of the 14th Meeting on the Mathematics of Language (MoL 2015)*, edited by Marco Kuhlmann, Makoto Kanazawa, and Gregory M. Kobele, 112–125. Chicago, USA.

Chandlee, Jane, and Jeffrey Heinz. Forthcoming. Strictly local phonological processes. *Linguistic Inquiry* .

Chandlee, Jane, Jeffrey Heinz, and Adam Jardine. To appear. Input strictly local opaque maps. *Phonology* .

Chomsky, Noam. 1956. Three models for the description of language. *IRE Transactions on Information Theory* 113124. IT-2.

Clark, Alexander, and Shalom Lappin. 2011. *Linguistic Nativism and the Poverty of the Stimulus*. Wiley-Blackwell.

Garcia, Pedro, Enrique Vidal, and José Oncina. 1990. Learning locally testable languages in the strict sense. In *Proceedings of the Workshop on Algorithmic Learning Theory*, 325–338.

Gold, E.M. 1967. Language identification in the limit. *Information and Control* 10:447–474.

Heinz, Jeffrey. 2010a. Learning long-distance phonotactics. *Linguistic Inquiry* 41:623–661.

Heinz, Jeffrey. 2010b. String extension learning. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, 897–906. Uppsala, Sweden: Association for Computational Linguistics.

Heinz, Jeffrey. 2016. Computational theories of learning and developmental psycholinguistics. In *The Oxford Handbook of Developmental Linguistics*, edited by Jeffrey Lidz, William Synder, and Joe Pater, chap. 27, 633–663. Oxford, UK: Oxford University Press.

Heinz, Jeffrey, Anna Kasprzik, and Timo Kötzing. 2012. Learning with lattice-structured hypothesis spaces. *Theoretical Computer Science* 457:111–127.

Heinz, Jeffrey, and James Rogers. 2013. Learning subregular classes of languages with factored deterministic automata. In *Proceedings of the 13th Meeting on the Mathematics of Language (MoL 13)*, edited by Andras Kornai and Marco Kuhlmann, 64–71. Sofia, Bulgaria: Association for Computational Linguistics.

de la Higuera, Colin. 2010. *Grammatical Inference: Learning Automata and Grammars*. Cambridge University Press.

Hopcroft, John E., and Jeffrey D. Ullman. 1979. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley.

Horning, J. J. 1969. A study of grammatical inference. Doctoral dissertation, Stanford University.

Jardine, Adam, Jane Chandlee, Rémi Eyraud, and Jeffrey Heinz. 2014. Very efficient learning of structured classes of subsequential functions from positive data. In *Proceedings of the Twelfth International Conference on Grammatical Inference (ICGI 2014)*, edited by Alexander Clark, Makoto Kanazawa, and Ryo Yoshinaka, vol. 34, 94–108. JMLR: Workshop and Conference Proceedings.

Kearns, Michael, and Umesh Vazirani. 1994. *An Introduction to Computational Learning Theory*. MIT Press.

McNaughton, Robert, and Seymour Papert. 1971. *Counter-Free Automata*. MIT Press.

Osherson, Daniel, Scott Weinstein, and Michael Stob. 1986. *Systems that Learn*. Cambridge, MA: MIT Press.

Rogers, Hartley. 1967. *Theory of Recursive Functions and Effective Computability*. McGraw Hill Book Company.

Rogers, James. 2003. wMSO theories as grammar formalisms. *Theoretical Computer Science* 293:291–320.

Rogers, James, Jeffrey Heinz, Gil Bailey, Matt Edlefsen, Molly Visscher, David Wellcome, and Sean Wibel. 2010. On languages piecewise testable in the strict sense. In *The Mathematics of Language*, edited by Christian Ebert, Gerhard Jäger, and Jens Michaelis, vol. 6149 of *Lecture Notes in Artifical Intelligence*, 255–265. Springer.

Rogers, James, Jeffrey Heinz, Margaret Fero, Jeremy Hurst, Dakotah Lambert, and Sean Wibel. 2013. Cognitive and sub-regular complexity. In *Formal Grammar*, edited by Glyn Morrill and Mark-Jan Nederhof, vol. 8036 of *Lecture Notes in Computer Science*, 90–108. Springer.

Rogers, James, and Geoffrey Pullum. 2011. Aural pattern recognition experiments and the subregular hierarchy. *Journal of Logic, Language and Information* 20:329–342.

Thomas, Wolfgang. 1982. Classifying regular events in symbolic logic. *Journal of Computer and Systems Sciences* 25:370–376.

Valiant, L.G. 1984. A theory of the learnable. *Communications of the ACM* 27:1134–1142.

Vidal, Enrique, Franck Thollard, Colin de la Higuera, Francisco Casacuberta, and Rafael C. Carrasco. 2005a. Probabilistic finite-state machines-part I. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 27:1013–1025.

Vidal, Enrique, Frank Thollard, Colin de la Higuera, Francisco Casacuberta, and Rafael C. Carrasco. 2005b. Probabilistic finite-state machines-part II. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 27:1026–1039.

Wexler, Kenneth, and Peter Culicover. 1980. *Formal Principles of Language Acquisition*. MIT Press.